High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA *

Yi-Hua E. Yang, Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California
{yeyang, prasanna}@usc.edu

ABSTRACT

We propose a *pipelined Dynamic Search Tree* (pDST) on FPGA which offers high throughput for LOOKUP, INSERT and DELETE operations as well as the capability to perform in-place incremental updates. Based on the pipelined 2-3 tree data structure, our pDST supports one LOOKUP per clock cycle and maintains tree balance under continual INSERT and DELETE operations. A novel *buffered update* scheme together with a *bi-directional linear pipeline* allows the pDST to perform one INSERT or DELETE operation per $O(\log N)$ cycles (N being the tree capacity) without stalling the LOOKUP operations. Nodes at each pipeline stage are allocated and freed by a *free-node chaining* mechanism which greatly simplifies the memory management circuit.

Our prototype implementation of a 15-level, 32-bit key dual-port pDST requires 192 blocks of 36 Kb BRAMs (64%) and 12.8k LUTs (6.3%) on a Virtex 5 LX330 FPGA. The circuit has a maximum capacity of 96k 32-bit keys and clock rate of 135 MHz, supporting 242 million LOOKUPs *and* concurrently 3.97 million INSERTs or DELETEs per second.

Categories and Subject Descriptors

B.5.1 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design—*Styles*; E.1 [**DATA STRUCTURES**]: *Trees*

General Terms

Algorithms, Design, Performance

Keywords

2-3 tree; B-tree; pipelined tree; dynamic update; incremental update; in-place update; IP routing; OpenFlow

*Supported by U.S. National Science Foundation under grant CCR-0702784

FPGA'10, February 21–23, 2010, Monterey, California, USA.

1. INTRODUCTION

Search trees are classic data structures fundamental to many application domains such as database, artificial intelligence, and data mining. Recent applications include DNA sequencing [7] and IP routing [3, 11, 10] where high throughput, easy construction and maintenance, and compact size of the data structure are required. A search tree attains its optimal performance when it is *balanced*, where the depth of the leaves are approximately equal.

A search tree can be either statically constructed or dynamically updated. The ability to perform dynamic updates is critical to high-performance and future network applications.[1] For example, an emerging class of network applications, such as those following the OpenFlow standard [1], requires more dynamic and frequent *flow* management. It was shown in [5] that total 2.8 million flows, over 90% of which shorter than a few minutes, were collected in a network trace during a 2-hour period. This corresponds to over 770 flow creations or teardowns per second. In [8] it was estimated that each Google data center could serve 100s of thousands of new flows per second. It would be very expensive, if at all feasible, to reconstruct an optimal search structure thousands of times per minute for timely updates.

An even more critical requirement of a good dynamic search tree (DST) is to continuously serve high throughput search operations under concurrent tree updates. This is especially true for network applications where the flow rate (inversely proportional to the packet *round-trip time*) is always much less than the packet rate (inversely proportional to the packet *transmission time*). For example, the 2-hour trace collected in [5] contained a total number of 168 million packets, 60 times higher than the number of flows.

To obtain optimal performance, a good DST also needs to maintain its tree balance efficiently after any number of update operations. Due to the structural changes needed to balance the tree, the DST can only perform one update (key INSERT or DELETE) operation at a time. On the other hand, concurrent search (key LOOKUP) operations *should* be possible if the tree nodes visited by the multiple searches are stored in different memory modules accessible in parallel.

Our goal in this study is to parallelize the common DST operations (LOOKUP, INSERT and DELETE) using a pipelined dynamic search tree on FPGA:

1. We design a *pipelined Dynamic Search Tree* (pDST) which accelerates the throughput of the LOOKUP oper-

[1] Internet routing utilize prefix match. Various schemes have been proposed to perform prefix match in a search tree [9, 10, 11]. Such schemes are out of the scope of this paper.

ations by $O(\log N)$ times, N being the size of the tree, compared with a software implementation.

2. We map the pDST onto a highly modular *bi-directional linear pipeline* (BLP) architecture on FPGA for streamlined LOOKUP, INSERT and DELETE operations.
3. We modify the original 2-3 tree algorithms and engineer the *buffered update* scheme to perform INSERT or DELETE operations in the BLP without blocking other LOOKUP operations.
4. We devise the *floating root* and *free-node chaining* mechanisms to grow and shrink the pDST without complex memory management.
5. We verify and evaluate our pDST design with a prototype implementation on FPGA.

Section 2 gives the background of our pipelined Dynamic Search Tree (pDST) design. The architecture of the pDST is explained in Section 3, while the algorithm engineering is detailed in Section 4. Section 5 evaluates our prototype implementation on FPGA. Section 6 describes the related work and Section 7 concludes the paper.

2. BACKGROUND

High throughput data queries and updates, as well as compact and scalable data storage are crucial to many high-performance applications. The search trees are an attractive data structure which provide deterministic performance and a rich set of operations [6].

A search tree is *dynamic* if it can be updated dynamically and incrementally. There are three basic operations for a dynamic search tree (DST): LOOKUP, INSERT and DELETE. The LOOKUP operation searches the tree for a particular key. The INSERT operation further adds the key into the tree if it was not found. The DELETE operation removes an existing key from the tree.

A DST offers optimal performance when it is balanced, *i.e.*, when all its leaf nodes have the same distance (or *depth*) from the root. Various DSTs, such as the AVL tree, Red-Black tree, 2-3 tree and B-tree, all maintain tree balance under INSERTs and DELETEs using per-level node operations. These node operations include *tree rotation* (for AVL and Red-Black), *node splitting* and *merging* (for 2-3 tree and B-tree). In software, these DSTs achieve $O(\log N)$ time complexity per operation with a capacity of N keys.

2.1 Motivation

In this work, we focus on the design of a hardware search tree architecture that is both dynamic and pipelined. Hardware acceleration is critical for applications that demand high performance. For example, it was reported in [4] that the hardware accelerated switch architecture achieves over 30x the throughput of the software version. The highly parallel and flexible logic and memory resources on FPGA make the platform an ideal choice for such designs.

We base the pipelined Dynamic Search Tree (pDST) architecture on the 2-3 tree data structure for two reasons. First, unlike AVL and Red-Black trees, there is no tree rotation involved when updating a 2-3 tree. It is fairly easy to perform tree rotations in software with centralized and sequential memory accesses; on the other hand, it is very difficult, if at all possible, to maintain the pipelined memory access in hardware after a sequence of tree rotations.

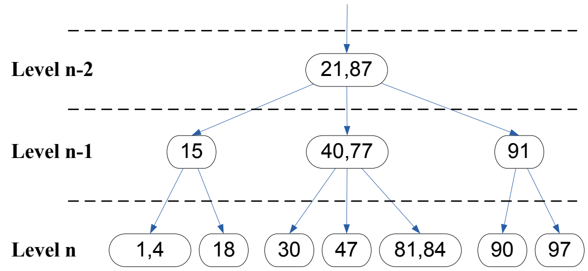

Figure 1: Example of a simple 2-3 tree.

Second, each node in a 2-3 tree is small enough (with one or two keys) so that node access and update are relatively simple. In contrast, the simplest B-tree (the 2-3-4 tree) has up to 3 keys and 4 child addresses per node, increasing memory bandwidth by 50% and circuit complexity over 2x compared to the 2-3 tree.

2.2 Basics of 2-3 Tree Data Structure

The 2-3 tree data structure is a variant from the B-tree family [6] and is shown related to the Left-Lean Red-Black tree data structure [12]. A node in a 2-3 tree can be either a *half node* (with 1 key and 2 child addresses) or a *full node* (with 2 keys and 3 child addresses). Each key in the node is sandwiched between two child addresses. Suppose keyA is sandwiched between addrL on the left and addrM on the right, then it is guaranteed that all keys stored in the subtree under addrL are smaller than keyA, while all those under addrM are greater. Figure 1 shows a simple 2-3 tree.

The LOOKUP operation in a 2-3 tree is similar to that in a basic binary search tree, except up to two key comparisons are performed per node. The INSERT and DELETE operations are more complicated and (non-tail) recursive in nature. The basic INSERT algorithm can be described as follows (adapted from [12]):

```
(Key, Addr) ⇐ INSERT(Key k, Addr p)
if p is null then {traverse past a tree leaf}
  return (k, null) {push key back to the leaf}
end if
if k is found in node[p] then
  abort {k already exists in tree}
end if
Find child addr p' to search for k recursively
(h, p') ← INSERT(k, p') {recursion}
if h is not null then {get a key from child}
  if node[p] is full then
    Got 3 keys and 4 addrs from h, p' and node[p]
    Create new node at address q
    Put left key & two addrs in node[p]
    Put right key & two addrs in node[q]
    Let m be the middle key
    return (m, q) {push key & addr to parent}
  else {node at addr p is half}
    Add h and p' into node[p]
    return (null, null)
  end if
end if
```

The DELETE operation is similar to but a bit more complex than INSERT. We will not detail the DELETE algorithm except the following major differences from INSERT:

1. While new keys are always inserted at the tree leaves, existing keys can be deleted from any internal node. When this happens, we must move the key's successor (which will always reside in a leaf node) to the (internal) node where the key is deleted from.

2. A node on the "DELETE path" can first conditionally *pull* a key from its parent, then conditionally *push* a key back to the parent, depending on whether the node and its neighbor is full or half.

Both these cases increase the number of cases to handle and the amount data to transfer for DELETE operations.

2.3 Challenges of pDST on FPGA

A balanced DST can perform each LOOKUP, INSERT or DELETE operation in $O(\log N)$ time, where N is the number of keys stored in the tree. There is however a major difference between the search (LOOKUP) and update (INSERT and DELETE) operations: the search operations do not change the tree structure and can be performed in parallel, while the update operations *can* change the tree structure and must be performed one at a time. Furthermore, to guarantee consistency and correctness, no search can be performed while the DST undergoes structural changes.

This posts a serious challenge to the pDST design in hardware. With a straightforward implementation, when an update operation trickles through the pipeline making structural changes to the DST, the entire pipeline needs to be stalled for up to $O(\log N)$ cycles. This greatly reduces the throughput performance and defeats the very purpose of pipelining. The problem is aggravated by the fact that the algorithms for both INSERT and DELETE in 2-3 tree are recursive in nature. This implies that the structural changes made by an update operation will even trickle through the tree levels in the reverse (bottom-up) direction. While it is possible to perform the structural changes on a copy of the tree before committing them in a single transaction [2], doing so requires a complex server-client system and defeats the purpose of hardware acceleration.

A perhaps more serious challenge is memory management. An arbitrary node per level can be allocated or freed by each INSERT or DELETE operation. For both simplicity and speed, the hardware circuit cannot afford a complex memory management scheme. On the other hand, we want the pDST to be able to utilize all available memory at every level. Thus a cheap but effective memory allocation mechanism is required for a practical pDST design on FPGA.

3. ARCHITECTURE DESIGN

We design a novel *bi-directional linear pipeline* (BLP) for our pipelined Dynamic Search Tree (pDST). Based on this architecture, we devise the *floating root*, *free-node chaining*, and *buffered update* mechanisms to efficiently maintain the balanced tree structure and perform non-blocking tree updates.

3.1 Bi-directional Linear Pipeline

The high-level structure of the bi-directional linear pipeline (BLP) is shown in Figure 2. The BLP really consists of two pipelines of opposite directions: a forward (downward) *search pipeline* and a backward (upward) *update pipeline*. All commands to the pDST are initially sent to the first stage (Stage 0) of BLP. Each command propagates through

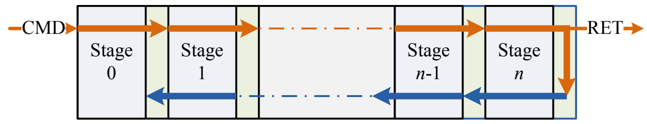

Figure 2: Overall structure of BLP.

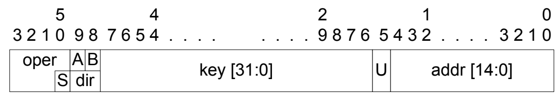

Figure 3: The pDST command format.

the search pipeline until reaching the last stage (Stage n), where it outputs a return value and is optionally relayed to the update pipeline towards the reverse direction.

3.1.1 Command Format and Processing

All commands follow the same linear flow throughout the BLP. A *command* represents an operation (LOOKUP, INSERT or DELETE) together with the associated data (key and address) and metadata (options). Figure 3 describes the command format. Each command is 54-bit wide, including 4 bits for the operation code, 32 bits for the key value, 15 bits for the per-stage node address, and 3 bits of options.

Originally, each command is constructed externally with one of the three operation codes (LOOKUP, INSERT or DELETE) and a *key* argument. All other fields are initialized to zeros. The content of the command fields will be modified from stage to stage throughout the BLP. The semantics of the fields depend on the operation context. Table 1 and Table 2 give an overview of the command field semantics under various scenarios. Details of the usage of these fields are discussed in Section 4. In particular, the S bit overlaps with the operation field and has the special meaning of "looking for successor." It is only set when a DELETE command finds the key to delete in an internal node and needs to find the key's successor.

Note that the width of the command can be different from that of Figure 3 if a different key length or tree capacity is used. Our key and address fields are 32 bits and 15 bits wide, respectively, because we had chosen a key length of 32 bits and a maximum tree height of 15 levels (excluding root). Also in the proof-of-concept design, we did not associate any output value with the keys.

3.1.2 Stage Architecture

All stages in BLP have a unified architecture whose simplified version is shown in Figure 4. The "heart" of a stage is the key comparator and the multiplexer control logic. These two units define the processing and flow of each operation.

Table 1: Semantics of LOOKUP and INSERT fields during search (S) and update (U).

Field	LOOKUP (S)	INSERT (S)	INSERT (U)
A	Matched keyA	Found keyA	(unused)
B	Matched keyB	Found keyB	(unused)
U	(unused)		Need to update
key	Key to search	Key to insert	Key to parent
addr	Next-stage node address		Addr. to parent

Table 2: Semantics of DELETE fields during various cycles of search (S) and update (U).

Field	DELETE (S1)	DELETE (S2)	DELETE (U1)	DELETE (U2)
A	Direction of node's neighbor	(unused)	(unused)	Type of processing
B			(unused)	
U	Parent is key match		Accept the successor	Propagate changes
key	Key to delete	Key of parent	Successor to deleted key	Key to parent
addr	Next-stage node addr.	Child's nbor. addr.	(unused)	Addr. to parent

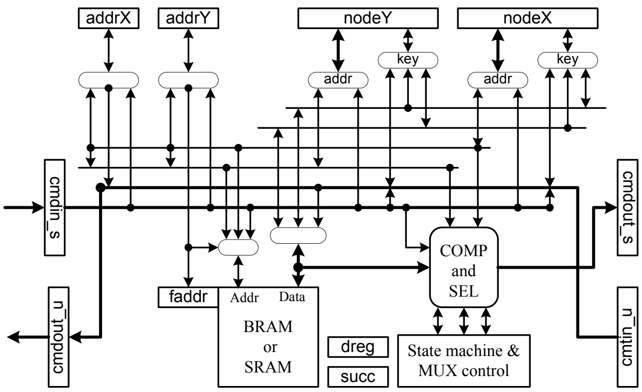

Figure 4: Architecture of a BLP stage.

The most complex part, however, is the switching fabric that connects the various circuit elements.

Although conceptually the search and update pipelines are separate, in practice they share accesses to most stage resources including the local memory and the various registers. This sharing is necessary to allow the INSERT and DELETE commands to be performed in a buffered and non-blocking (with respect to LOOKUPs) manner (see Section 3.3). The LOOKUP commands, on the other hand, are rather simple and do not access all the stage resources other than the key comparator and local memory.

Every stage performs the same processing for each command; the last stage (leaf level) also needs to optionally relay a INSERT of DELETE command from the search pipeline to the update pipeline:

Leaf-level operations for INSERT *commands*

- If the leaf node was a half node:
  - Insert the key into the leaf node.
  - Relay an empty command to the update pipeline.
- If the leaf node was a full node:
  - Relay the insert key command with a null address (see Algorithm 3) to the update pipeline.

Leaf-level operations for DELETE *commands*

- If the key to delete was found in an internal node:
  - Remove the key's successor from the leaf node.
  - Relay a "send successor" command (see Algorithm 7) with the successor key to the update pipeline.
  - If the leaf node was half, then also relay a "children merged" command (see Algorithm 8) to the update pipeline.
- If the key to delete is found in a leaf node:
  - Remove the key-to-delete from the node.
  - If the node was full, then an empty command is relayed to the update pipeline.
  - If the node was half, then a "children merged" command (see Algorithm 8) is relayed to the update pipeline.

Table 3: Stage registers used by LOOKUP (L), INSERT (N) and DELETE (D) commands.

Name	Description	Used by
depth	Depth of the current stage	L,N,D
cmdin_s	Input command, search pipeline	L,N,D
cmdout_s	Output command, search pipeline	L,N,D
cmdin_u	Input command, update pipeline	N,D
cmdout_u	Output command, update pipeline	N,D
mem[·]	Stage-local memory access	L,N,D
addrX	Address reg. X for memory update	N,D
addrY	Address reg. Y for memory update	N,D
nodeX	Node reg. X for memory update	N,D
nodeY	Node reg. Y for memory update	N,D
faddr	Next free memory address	N,D
dreg	Direction register (left,middle,right)	N,D
succ	Wait for key successor (binary)	D

These additional steps make the control logic of the last stage slightly more complex. On the other hand, since the leaf stage does not need to handle child address, its switching fabric can be greatly simplified.

Table 3 lists all the stage registers, their description and usage. The depth value is a special constant storing the stage's depth within the pipeline. It is coded in the state machine to control the scheduling of memory accesses by the INSERT and DELETE commands in the update pipeline. The *cmdin*'s and *cmdout*'s are 54-bit command buffers described in Figure 3. Each stage has two sets of command buffers, one for the search pipeline and the other for the update pipeline.

The *addrX*, *addrY* and *faddr* are 15-bit address registers. The *nodeX* and *nodeY* are 128-bit node registers in the format of Figure 5. The *dreg* and *succ* registers are special flags which support the INSERT and DELETE operations.

3.1.3 Floating Root

A 2-3 tree always adds and removes new nodes at the bottom (leaf) of the tree, and propagates the required structural changes, if any, towards the tree root. Thus the tree height can be seen as growing and shrinking on the top (root) of the tree. We design the BLP to populate from the bottom (Stage n) up in a similar manner. This makes the root of the pDST "floating" across the BLP pipeline amid continual INSERT and DELETE commands. Initially a single (root) node containing one or two keys is placed at Stage n. At

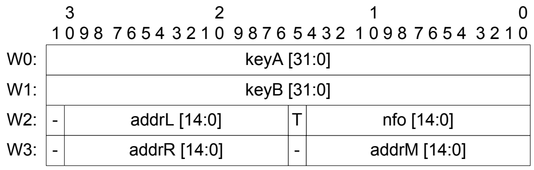

Figure 5: Format of the node data.

this point, all the Stages 0 through n-1 are *inactive* in that their local memories do not hold any valid pDST node. An inactive stage simply passes along any command and does not do any processing.

The root will implicitly "float" from Stage i to Stage i-1 when a key is popped up from Stage i along the update pipeline to Stage i-1 due to some INSERT command. A special root address register is updated with the stage number and stage-local node address of the current root, right in front of Stage 0. Similarly, the root can float from Stage i to Stage i+1 when the last node at Stage i is freed due to some delete command, which also updates the root address register accordingly. The floating root mechanism guarantees that all inactive stages are "above" the pDST root, *i.e.*, they must be packed at the upper-most part of the BLP.

3.2 Memory Management

Figure 5 shows the data format of a single node in the memory. A 128-bit memory interface is used to access all four 32-bit words (W0 – W3) in one cycle. Each node consists of 2 keys (keyA and keyB), 3 child addresses (addrL, addrM and addrR), plus a "next free offset" (nfo) initialized to 0. A 1-bit "type" flag (T) indicates whether the node is full or half. The node update algorithms ensure that

1. keyA < keyB
2. addrL leads to keys less than keyA
3. addrR leads to keys greater than keyB
4. addrM leads to keys between keyA and keyB

If a node is full (T=1), then both keys and all 3 addresses are used. Otherwise, only keyA, addrL and addrM are used.

The pDST hardware does not offer sophisticated memory management which can greatly increase circuit complexity and slow down memory access. Instead, a novel *free-node chaining* mechanism is used to efficiently allocate and free nodes as requested by the tree updates:

- Initially, the nodes are allocated in the stage local memory in a linear fashion, starting from the lowest address (0x1, since 0x0 is reserved for the invalid address). The *faddr* stage register stores the next free address to allocate.
- After a new node ($nodeX$) is allocated at the address pointed to by *faddr*, the value of *faddr* is updated with ($faddr + nodeX$.nfo + 1).
- When an existing node ($nodeY$) at address $addrY$ is to be freed, $nodeY$.nfo is first set to ($faddr - addrY - 1$), then *faddr* is updated with $addrY$.

Using the free-node chaining mechanism, each stage forms an implicit chain of free nodes in its local memory headed by the *faddr* register. The cost is only one extra address field (nfo) per node. Furthermore, since the nfo field is *only* used for "next free offset" when the node itself is *un*used, the field can be overloaded with other purposes (such as key output) when the node *is* used for key storage. While not exploited in our proof-of-concept implementation, such application-specific optimization could eliminate the only overhead required by the free-node chaining mechanism.

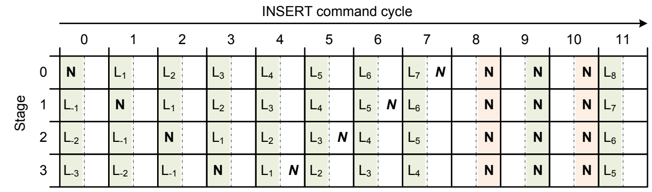

Figure 6: BLP scheduling of the insert command.

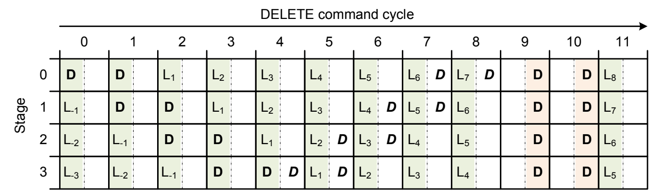

Figure 7: BLP scheduling of the delete command.

3.3 Buffered Update

We devise the buffered update mechanism allowing non-blocking INSERT and DELETE commands (with respect to LOOKUPs) by ensuring the following properties of the BLP:

1. At any time there is at most one INSERT or DELETE in the entire pipeline.
2. Each stage (either the search or the update part) performs at most one memory access per cycle.
3. Other than the command buffers, all other stage registers are accessed only by INSERTs and DELETEs.

The buffered update mechanism works as follows. An INSERT (or DELETE) command first goes down the search pipeline to collect node (and neighbor) information. The command makes a memory read at every stage in the search (forward) process, where the read data are stored in the stage registers. At the leaf-level stage, the command is relayed to the update pipeline, where it propagates any tree changes in the reverse direction. The changes are made to the data buffered in the stage registers *without* accessing the stage memory. After the command propagates back to Stage 0, the entire search pipeline is stalled for 2 (for DELETE) or 3 (for INSERT) cycles, when the update pipeline commits the buffered changes to the local memory at every stage.

Figure 6 and Figure 7 show the scheduling of INSERT (N) and DELETE (D) concurrently with multiple LOOKUPs (L_{-3} to L_8) on a hypothetical 3-level pDST. Every solid block, divided into two halves by the dotted line, represents a stage-cycle. The left half represents the search (forward) pipeline and the right half the update (backward) pipeline. A shaded half-block represents a memory read (green) or write (red) at the stage. Italic font represents command propagation in the reverse (backward) direction.

Note that it is possible for an INSERT or DELETE command to modify some node currently accessed by a LOOKUP. For example in Figure 6, a node at Stage 2 can first be read by L_5 in cycle 7, then modified by the INSERT command in cycle 8. When this occurs, we simply mark the conflicting LOOKUP command (L_5 in the example) as *invalid* and let it propagate through the rest of the pipeline without further processing. A higher level mechanism can be used to reschedule such a LOOKUP command.[2]

Each LOOKUP or INSERT command occupies only one slot in the search pipeline, while a DELETE command occupies two. For each INSERT, a stage spends one memory-read cycle in the search pipeline, plus one memory-read and two memory-write cycles in the update pipeline. For each DELETE, the stage spends two memory-read cycles in the search pipeline and two memory-write cycles in the update pipeline. All 3 properties discussed in the beginning of the section are thus observed.

In a pDST with k levels (where the corresponding BLP spans from Stage 0 to Stage k), the maximum command rate R per cycle for INSERT and DELETE can be calculated as

$$R_{\{\mathrm{N,D}\}} = \frac{1}{(2k+3)} \tag{1}$$

The command rate of LOOKUP under the maximum INSERT and DELETE rate will be

$$R_{\mathrm{L}} \;=\; 1 - 4 \times R_{\{\mathrm{N,D}\}} = \frac{(2k-1)}{(2k+3)} \tag{2}$$

The overall command rate of the pDST with the maximum update capability is

$$R = R_{\{\mathrm{N,D}\}} + R_{\mathrm{L}} = \frac{2k}{2k+3} \tag{3}$$

Effectively, each LOOKUP command takes one cycle per stage, whereas each INSERT/DELETE takes 4. The overall command rate is just slightly less than one per cycle $\log N$ times higher than the original 2-3 tree software implementation where N is the tree capacity.

4. ALGORITHM ENGINEERING

We modify the original 2-3 tree algorithms for pDST so that the LOOKUP, INSERT and DELETE operations performed in a pipelined and non-blocking fashion. Since each stage in the BLP may spend multiple cycles on the INSERT and DELETE commands, our algorithms incorporate the concept of *cycle*. We start the cycle count (from 0) of a command at the time it is received by Stage 0. In the search pipeline, each command is always processed by Stage i in cycle i of the command. In addition, during the stalls where the buffered changes are committed by the update pipeline, the cycle counts of the LOOKUP commands in the search pipeline do not increase.

4.1 The LOOKUP Command

Algorithm 1 describes the per-stage processing for the LOOKUP command. If the key has been found (line 3), then the command is simply passed to the next stage. Otherwise, a node is read from the address specified by the previous stage (line 6) and compared with the key in the command. The corresponding child address is then forwarded along the command (*cmdout_s*) to the next stage. Note that *ntmp* is defined (in line 6) merely as a variable representing the memory output, not a physical register.

[2] An alternative way is to write the tree updates to stage memory using command bubbles that propagate through the search pipeline. This however could increase circuit complexity and lengthen the INSERT/DELETE latency.

Algorithm 1 Per-stage LOOKUP processing, search pipeline.

1: **BEGIN** cycle depth
2: *cmdout_s* $\leftarrow$ *cmdin_s*
3: **if** *cmdin_s*.A $= 1$ or *cmdin_s*.B $= 1$ **then**
4: goto line 28
5: **end if**
6: *ntmp* := mem[*cmdin_s*.addr]
7: **if** *ntmp*.T $= 0$ **then** {current node is half}
8: **if** *cmdin_s*.key $<$ *ntmp*.keyA **then**
9: *cmdout_s*.addr $\leftarrow$ *ntmp*.addrL
10: **else if** *cmdin_s*.key $=$ *ntmp*.keyA **then**
11: *cmdout_s*.A $\leftarrow 1$ {matching keyA}
12: **else** {*cmdin_s*.key $>$ *ntmp*.keyA}
13: *cmdout_s*.addr $\leftarrow$ *ntmp*.addrM
14: **end if** {finish compare half node}
15: **else** {*ntmp*.T $= 1$, current node is full}
16: **if** *cmdin_s*.key $<$ *ntmp*.keyA **then**
17: *cmdout_s*.addr $\leftarrow$ *ntmp*.addrL
18: **else if** *cmdin_s*.key $=$ *ntmp*.keyA **then**
19: *cmdout_s*.A $\leftarrow 1$ {matching keyA}
20: **else if** *cmdin_s*.key $<$ *ntmp*.keyB **then**
21: *cmdout_s*.addr $\leftarrow$ *ntmp*.addrM
22: **else if** *cmdin_s*.key $=$ *ntmp*.keyB **then**
23: *cmdout_s*.B $\leftarrow 1$ {matching keyB}
24: **else** {*cmdin_s*.key $>$ *ntmp*.keyB}
25: *cmdout_s*.addr $\leftarrow$ *ntmp*.addrR
26: **end if** {finish compare full node}
27: **end if**
28: **END** cycle depth

While the LOOKUP algorithm is fairly simple and straightforward, it serves as the backbone for both INSERT and DELETE in the search pipeline. Specifically, determining the type of the node (line 7 and 15) and finding the child address for the next stage are fundamental to the pDST traversal.

4.2 The INSERT Command

The INSERT algorithm for the search pipeline (Algorithm 2) follows the same structure as LOOKUP, except that the node address and node data are now stored in the stage registers (Algorithm 2 line 6 and 7) to be used for the update processing later. In addition, the direction towards which the child address is found is also recorded (see the descriptions in Algorithm 2 line 9 and line 11). This "child direction" (*dreg*) helps saving a comparison during the tree update (Algorithm 3) when a key is popped up from the child level.

Algorithm 3 describes the processing performed at each stage to propagate an INSERT command through the update pipeline. For simplicity and due to space limitation, we do not separately detail the processing performed by the leaf level, whose differences from the internal levels have been explained in Section 3.1.2.

An INSERT command in the update pipeline can either contain a valid {key, address} pair to the upper level, or contain nothing and be ignored (Algorithm 3 line 3). When a {key, address} pair is received, it is either added to a half node (Algorithm 3 lines 6 to 10), or induces a node split in a full node and sends another {key, address} pair to the next upper level (Algorithm 3 lines 11 to 25).

Algorithm 2 Per-stage INSERT processing, search pipeline.

```
 1: BEGIN cycle depth
 2: cmdout_s ← cmdin_s
 3: if cmdin_s.A = 1 or cmdin_s.B = 1 then
 4:    goto line 13
 5: end if
 6: addrX ← cmdin_s.addr
 7: nodeX ← mem[cmdin_s.addr]
 8: if nodeX.T = 0 then {current node is half}
 9:    (Similar to Algorithm 1 lines 8 to 14, except dreg is
       set to left or middle dep. on the comparison)
10: else {nodeX.T = 1, current node is full}
11:    (Similar to Algorithm 1 lines 16 to 26, except dreg is
       set to left, middle or right dep. on the comparison)
12: end if
13: END cycle depth
```

Algorithm 3 Per-stage INSERT proc., update pipeline 1/2.

```
 1: BEGIN cycle (31-depth)
 2: cmdout_u ← cmdin_u
 3: if cmdin_u.U = 0 then
 4:    goto line 27
 5: end if
 6: if nodeX.T = 0 then
 7:    addrY ← null {not to create new node}
 8:    (depending on the value of dreg, shuffle cmdin_u.key
       & cmdin_u.addr into nodeX)
 9:    nodeX.T ← 1 {nodeX become full}
10:    cmdout_u.U ← 0 {pop no key to parent}
11: else {nodeX.T = 1}
12:    addrY ← faddr {create new node at faddr}
13:    cmdout_u.addr ← faddr
14:    (split nodeX to nodeY with proper child addrs)
15:    nodeX.T ← 0, nodeY.T ← 0
16:    if dreg = left then
17:       cmdout_u.key ← nodeX.keyA
18:       nodeX.keyA ← cmdin_u.key
19:    else if dreg = middle then
20:       cmdout_u.key ← cmdin_u.key
21:    else {dreg = right}
22:       cmdout_u.key ← nodeY.keyA
23:       nodeY.keyA ← cmdin_u.key
24:    end if
25:    cmdout_u.U ← 1 {pop key,addr to parent}
26: end if
27: END cycle (31-depth)
```

Algorithm 4 Per-stage INSERT proc. update pipeline 2/2.

```
 1: BEGIN cycle 32
 2: mem[addrX] ← nodeX
 3: if addrY = null then
 4:    goto line 12
 5: end if
 6: END cycle 32
 7: BEGIN cycles 33
 8: faddr ← addrY+1+mem[addrY].nfo
 9: END cycle 33
10: BEGIN cycles 34
11: mem[addrY] ← nodeY
12: END cycles 34
```

The node splitting is done similar to the original 2-3 tree, where the full node together with the incoming {key, address} pair would produce two half nodes and an extra (middle) key. The *dreg* register assists the hardware to determine where the incoming {key, address} should go and how to split the nodes (Algorithm 3 lines 16, 19 and 21). Due to space limitation, we do not detail all operations that are similar to the "half node" cases.

For the pDST of 15 levels, it is guaranteed that after 32 cycles the insert command would have propagated back to Stage 0. At this time all the changes to the pDST have been stored in the stage registers at the respective stages. In cycle 32 through 34, all stages will temporarily stall the search pipeline processing (of LOOKUP commands) to commit the changes required by the INSERT to memory.

Up to two memory writes per stage are needed to commit the changes: one for the node on the "INSERT path," (Algorithm 3 line 2), and optionally another for the node newly created (Algorithm 3 line 11). The new node to be allocated needs to be first read from memory to get its "next free offset" (nfo) field (Algorithm 3 line 8). This is required by the free-node chaining mechanism to update the *faddr* register.

4.3 The DELETE Command

The DELETE operation is considerably more complicated than both LOOKUP and INSERT for reasons similar to that described in Section 2.2. In the search pipeline alone, each DELETE operation requires two command words to be issued back-to-back in two cycles (see also the scheduling of the DELETE command in Figure 7).

Algorithm 5 Per-stage DELETE proc., search pipeline 1/2.

```
 1: BEGIN cycle depth
 2: cmdout_s ← cmdin_s
 3: cmdout_s.U ← 0 {assume no key match}
 4: succ ← cmdin_s.U {parent key match?}
 5: addrX ←cmdin_s.addr
 6: nodeX ← mem[cmdin_s.addr]
 7: if nodeX.T = 0 then
 8:    if cmdin_s.key = nodeX.keyA then
 9:       cmdout_s.S ← 1 {find key successor}
10:       cmdout_s.U ← 1
11:    else
12:       cmdout_s.addr ← (child address)
13:       cmdout_s.dir ← (child's nbor. direction)
14:       addrY ← (child's nbor. address)
15:    end if {finish compare half node}
16:    dreg ← cmdin_s.dir {nodeX's nbor. dir.}
17: else {nodeX.T = 1}
18:    (similar to lines 8 to 15, except compare both
       nodeX.keyA and nodeX.keyB)
19:    if cmdin_s.key < nodeX.keyB then
20:       dreg ← left {pull nodeX.keyA to child}
21:    else {cmdin_s.key ≥ nodeX.keyB}
22:       dreg ← right {pull nodeX.keyB to child}
23:    end if
24: end if
25: END cycle depth
```

From a high-level point of view, two command words are required because the DELETE operation requires the knowledge of both the node on the DELETE path and the node's neighbor in order to update the node structures. These up-

Algorithm 6 Per-stage DELETE proc., search pipeline 2/2.

```
 1: BEGIN cycle (depth+1)
 2: cmdout_s.addr ← addrY {send child's nbor.}
 3: if nodeX.T = 0 then {current node is half}
 4:    cmdout_s.key ← nodeX.keyA
 5:    if succ = 0 then
 6:       nodeX.keyA ← cmdin_s.key
 7:    end if
 8:    addrY ← cmdin_s.addr {neighbor address}
 9:    nodeY ← mem[addrY] {neighbor node}
10: else {nodeX.T = 1, current node is full}
11:    if dreg = left then {move keyA}
12:       cmdout_s.key ← nodeX.keyA
13:    else {dreg = right, move keyB}
14:       cmdout_s.key ← nodeX.keyB
15:    end if
16:    addrY ← null {no neighbor access}
17: end if
18: END cycle (depth+1)
```

Algorithm 7 Per-stage DELETE proc., update pipeline 1/2.

```
 1: BEGIN cycle (31-depth)
 2: cmdout_u ← cmdin_u
 3: if cmdin_u.U = 1 & nodeX.T = 0 & nodeY.T = 0 then
 4:    cmdout_u.U ← 1 {change beyond parent}
 5: else
 6:    cmdout_u.U ← 0 {change stops at parent}
 7: end if
 8: if succ = 1 then {node is waiting for successor}
 9:    if cmdin_u.U = 0 | nodeX.T = 1 then
10:       cmdout_u.S ← 1 {successor to parent}
11:    else {pull successor from parent}
12:       nodeX.keyA ← cmdin_u.key
13:    end if
14: end if
15: if cmdin_u.S = 1 then {accept successor}
16:    cmdout_u.S ← 0
17:    (insert cmdin_u.key to nodeX)
18:    dreg ← middle {update with successor}
19: end if {}
20: END cycle (31-depth)
```

dates could be: (1) remove a key from the current node only, (2) move a parent key to replace a key in the current node and send a neighbor's key to the parent, or (3) move a parent key to replace a key in the current node before merging with the node's neighbor.

The first command word (Algorithm 5) is responsible of comparing the keys in the current node with the key to delete. In the process, it will find and optionally store four pieces of information: (1) whether the key-to-delete is found in the current node, where the key successor must be searched; (2) the child node's address on the delete path; (3) the child's neighbor address and direction; and (4) the value of a key which will be potentially pulled to the child stage. Note that all four pieces of information can be obtained by one memory read (line 6) and no more than two key comparisons. If the current node is a half node, then it will also remember its neighbor's direction (in *dreg*) passed from its parent (line 16); otherwise (if the current node is full), it will store the "side" of the potentially pulled key (lines 20 and 22), which will be retrieved and carried to the child stage by the second command word.

The second command word (Algorithm 6) will optionally carry the *child's neighbor address* (line 2) and the *potentially pulled key* (line 4, 12 or 14) found by the first command word to the child stage. At the child stage, the neighbor will be read from stage memory to *nodeY* while the potentially pulled key stored in *nodeX* if and only if *nodeX* is a half node (lines 9 and 6, respectively). This is to prepare for the possible scenario where the (half) *nodeX* needs to pull a key from its parent and merge with its neighbor later (see discussion near the end of Section 2.2).

In addition, a complication arises in Algorithm 5 when the key to delete is found in an internal node. When this occurs, the command is marked with the "find successor" flag and the immediate next stage is notified (Algorithm 5 lines 9 and 10). The notification of the parent key match, when received (Algorithm 5 line 4), informs the stage *not* to accept the deleted key (*i.e.*, *succ* = 1 and Algorithm 6 line 6 will be skipped).

The DELETE operation also requires two back-to-back command words for the update process, whose algorithms are described in 2 parts. Algorithm 7 describes the per-stage processing of the first command word. Depending on whether a successor is found (see discussion in Section 3.1.2), the first DELETE command word carries the successor to either the internal node where the key was deleted (line 15 to 19), or the node's child which would "pull" the successor from the parent (line 12).

Algorithm 8 describes the processing of the second command word of DELETE in the update pipeline, as well as the final two memory write cycles. Although the algorithm is long, its concept is quite simple. Each stage in the update pipeline is met with one of three scenarios:

1. The child node and its neighbor merged and pulled a key from the current node (Algorithm 8 line 4).
2. The child's neighbor was full and sent up a key to the current node (Algorithm 8 line 19).
3. No change was propagated from the child stage to the current stage (Algorithm 8 line 26).

If case 3. above was the case, then nothing more needs to be done by this and all upper stages. If case 2. above was the case, then the key sent from the child level is added to the current node on the DELETE path, and no more processing needs to be performed by the upper stages.

Case 1. is further divided into three sub-cases:

1. The current node on the DELETE path was full (line 5). The node loses a key to its child and becomes a half node. No further processing propagates to the upper stages.
2. The node on the DELETE path was half, but its neighbor was full (line 8). The node loses a key to its child and obtains a key from its parent. The neighbor sends a key to the parent and becomes a half node.
3. Both the node on the DELETE path and its neighbor are half (line 12). The node loses a key to its child and merges the key it obtains from its parent with the neighbor.

After the second command word propagates back to Stage 0 (at cycle 32), both search and update pipelines will be stalled for two cycles (cycle 33 and cycle 34) where the buffered changes of the DELETE operation are committed to the local memory at every stage. The steps are similar to those for the insert command, except here the reverse direction of the free-node chaining mechanism is performed, and there is no need to read the "next free offset" field of the node (*nodeY*.nfo) since it has been obtained with *nodeY* during the search pipeline processing.

Algorithm 8 Per-stage DELETE proc., update pipeline 2/2.

```
BEGIN cycle (32-depth)
cmdout_u ← cmdin_u
cmdout_u.[A,B] ← [0,0]
if cmdin_u.A = 1 then {children merged}
  if nodeX.T = 1 then {current node is full}
    (depending on dreg, remove either keyA or keyB
    from nodeX)
    nodeX.T ← 0 {changed node to half}
  else if nodeY.T = 1 then {node half, nbor. full}
    cmdout_u.B ← 1 {send neighbor key}
    (depending on dreg, put either keyA or keyB of
    nodeY into cmdout_u.key)
    nodeY.T ← 0 {change nbor. to half}
  else {node & nbor. are half}
    cmdout_u.A ← 1 {nodes merged}
    cmdout_u.addr ← addrX
    (depending on dreg, merge nodeY to the left or right
    side of nodeX)
    nodeX.T ← 1 {change node to full}
    nodeY.T ← 1 {free neighbor}
  end if
else if cmdin_u.B = 1 then {key from child's nbor.}
  if nodeX.T = 0 or dreg = left then
    nodeX.keyA ← cmdin_u.key
  else {nodeX.T = 1 and dreg = right}
    nodeX.keyB ← cmdin_u.key
  end if
  addrY ← null {no neighbor access}
else {no update}
  if dreg ≠ middle then {not write successor}
    addrX ← null
  end if
  addrY ← null {not access neighbor}
end if
END cycle (32-depth)

BEGIN cycle 33
if addrX ≠ null then
  mem[addrX] ← nodeX
end if
END cycle 33
BEGIN cycle 34
if addrY ≠ null then
  if nodeY.T = 1 then {free nodeY}
    nodeY.nfo ← faddr-addrY-1
    faddr ← addrY
  end if
  mem[addrY] ← nodeY
end if
END cycle 34
```

5. IMPLEMENTATION & EVALUATION

We implemented a single stage of the bi-directional linear pipeline (BLP) in VHDL and verified its functional correctness against the algorithms described in Section 4. The stage was duplicated 15 times, plus a specially designed leaf stage circuit, to form the 15-level pipelined Dynamic Search Tree (pDST). We target our implementation on the Xilinx Virtex 5 LX330 FPGA. Table 4, 5, and 6 show the resource usage of the 15-level pDST prototype.

The stage circuit was written in behavioral VHDL. The complexity of the circuit is comparable to a simple pipelined SIMD processor with only 3 instructions (LOOKUP, INSERT, and DELETE). The circuit has a rather long 7.4 ns clock period, or a 135 MHz clock rate, due mainly to the complex DELETE processing and the lack of hand-tuned optimization in our prototype implementation. However, by pipelining along the tree depth and utilizing a separate memory for each tree level, the pDST can achieve an *effective latency* of one search per cycle. To utilize the dual-port capability of the on-chip memory, the BLP includes a second search-only pipeline. The added complexity was small since it only supports the LOOKUP commands. The resulting dual-ported BLP achieves a LOOKUP throughput of **242 Mop/s** *plus* an INSERT or DELETE throughput of **3.97 Mop/s**. This level of performance (∼4.1 ns per LOOKUP and ∼250 ns per INSERT/DELETE) is unparalleled by any software-based implementations currently available.

As shown in Table 4, the stage circuit is physically divided into 4 parts:

- CMP — The key comparator matching two pairs of 32-bit values in parallel and use the result to select one of the three 15-bit address inputs.
- BUF — The stage registers including their respective input multiplexers.
- CTRL — The state machine and cycle counters controlling the multiplexers and the INSERT and DELETE schedules.
- MISC — The total LUT usage minus the sum of the previous 3 parts. They include the memory addressing logic, the pipeline registers (for command buffering), and the switching fabric, etc.

Because each BRAM block consists of at least 1024 addressable words (either 18-bit or 36-bit wide depending on the configuration), we implemented the first 9 stages in LUT-based distributed memory (distRAM), as shown in Table 5. We assign 2^i nodes to level i of the pDST, $0 \le i \le 15$. Since a node at level i may have up to 3 children at level $i+1$, not all the capacity of our pDST prototype can be utilized. The maximum wastage in our implementation can be estimated as

$$2^{15} \times \left(\sum_{j=0}^{15} \frac{1}{2^j} - \sum_{k=0}^{10} \frac{1}{3^k} \right) \approx 2^{15} \times (2 - 1.5) \approx 16k \text{ nodes}$$

Thus our pDST prototype would have a maximum capacity of $48k$ nodes or about **96*k* 32-bit keys**. In practice, the optimal size ratio between consecutive pDST levels should be somewhere between 2 to 3, depending on the dynamics of the tree growth. We note that because the size of a balanced tree increases exponentially with respect to the tree

Table 4: XC5VLX330 6-LUTs for circuit logic

Func.	CMP	BUF	CTRL	MISC	Total
# LUTs	2741	4176	798	5394	12.8k (6.3%)

Table 5: XC5VLX330 6-LUTs for key storage

Stage	4	5	6	7	8	9	Total
# nodes	16	32	64	128	256	512	510
# LUTs	44	48	96	200	416	864	1668 (0.8%)

Table 6: XC5VLX330 RAMB36s for key storage

Stage	10	11	12	13	14	15	Total
# nodes	1k	2k	4k	8k	16k	32k	65024
# blocks	4	8	16	32	64	96	224 (78%)

height, the capacity of the pDST can be increased dramatically with only a few stages of external memory accesses, subject to the pin limitation of the FPGA.

The last stage (Stage 15) uses less BRAM than twice of the previous stage because the leaf nodes do not need to store the child addresses. This allows us to reduce the width of the memory from 144 bits (4×36 bit BRAM blocks) down to 108 bits (3×36 bit BRAM blocks). To enable the true dual-port capability, we assign BRAM widths in multiples of 36 bits. This incurs additional memory wastage whose optimization is not considered in our prototype implementation.

6. RELATED WORK

Dynamic search tree (DST) data structures have been known for a long time [6]. Some application-specific variation of the classical DSTs were proposed recently for the IP routing application [11, 3]. Both are software-based approaches, with [11] focusing on update performance and [3] on memory efficiency.

A distributed B-tree implemented over multiple servers and clients was implemented in [2]. It allows parallel search and update operations by maintaining a lazy copy of the (partial) tree at each client, which first calculates the changes locally then commits them in a single transaction. It also supports load balancing and hot-plugging of the servers.

Hardware-based binary search tree for IP lookup was proposed in [9]. To the best of our knowledge, however, our pDST is the first work that addresses both pipelining and dynamic updates efficiently in hardware.

7. CONCLUSION AND FUTURE WORK

Our pipelined Dynamic Search Tree (pDST) achieves both high update throughput *and* pipelined search performance. The pDST INSERT and DELETE algorithms pipeline the 2-3 tree in a linear fashion while observing no more than one memory access per stage at any time.

The pDST is mapped to a bi-directional linear pipeline (BLP) where we employ the novel techniques of floating root, free-node chaining, and buffered update. Our prototype implementation shows that the BLP architecture is practical for FPGA implementations and can further benefit from application-specific optimization.

With proper modifications, the pDST architecture can be adapted for various high-performance applications such as packet flow processing, pattern matching, DNA sequencing and data mining.

8. REFERENCES

[1] OpenFlow. http://www.openflowswitch.org, 2008.

[2] Aguilera, M., Golab, W., and Shah, M. A Practical Scalable Distributed B-Tree . In *34th Intl. Conf. on Very Large Data Bases* (2008).

[3] Behdadfar, M., Saidi, H., Alaei, H., and Samari, B. Scalar Prefix Search: A New Route Lookup Algorithm for Next Generation Internet. In *IEEE INFOCOM* (2009).

[4] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking Control of the Enterprise. In *ACM SIGCOMM* (2007).

[5] chan Lan, K., and Heidemann, J. A Measurement Study of Correlation of Internet Flow Characteristics. *Computer Networks 50*, 1 (January 2006), 46–62.

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[7] Hunt, E., Atkinson, M. P., and Irving, R. W. Database indexing for large DNA and protein sequence collections. *The VLDB Journal 11*, 3 (2002), 256–271.

[8] Joseph, D. A., Tavakoli, A., and Stoica, I. A Policy-aware Switching Layer for Data Centers. In *ACM SIGCOMM* (2008).

[9] Le, H., and Prasanna, V. Scalable High Throughput and Power Efficient IP-Lookup on FPGA. In *Proc. of 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2009).

[10] Lu, H., and Sahni, S. A B-Tree Dynamic Router-Table Design. *IEEE Trans. Comput. 54*, 7 (2005), 813–824.

[11] Sahni, S., and Lu, H. Dynamic Tree Bitmap for IP Lookup and Update. In *International Conference on Networking* (2007).

[12] Sedgewick, R. Left-Leaning Red-Black Tree. http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf, September 2008.