CSE 332: Data Structures and Parallelism

Spring 2022
Richard Anderson
Lecture 6: Binary Heaps and Algorithm Analysis

Announcements

- Reading: Weiss
  - Friday: Algorithm Analysis, 2.1-2.2
  - Monday: Binary Search Trees, 4.1-4.3, 4.6
- Project #1 Due Tuesday
  - Write up portion is important
    - Linked handout provides guidance
  - Run zip experiment and fuzzy testing to reveal bugs

Today

- Heap Initialization
  - Put n elements into a heap in O(n) time
- Algorithm analysis for recursive programs

Building a Heap

12	5	11	3	10	6	9	4	8	1	7	2

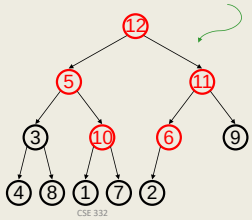
BuildHeap: Floyd's Method

12	5	11	3	10	6	9	4	8	1	7	2

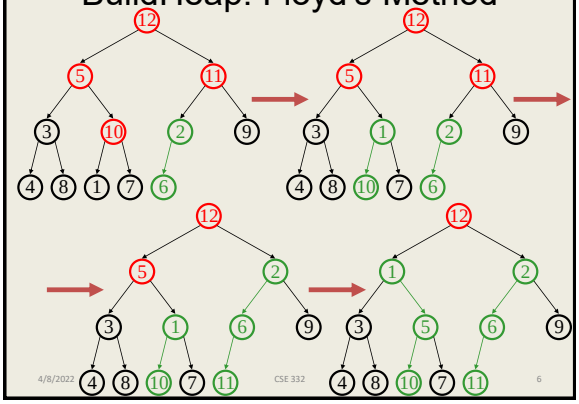
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!

Red nodes need to percolate down

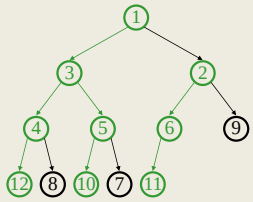
Key idea: fix red nodes from **bottom-up**



BuildHeap: Floyd's Method



Finally . . .



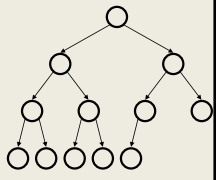
Buildheap pseudocode

```
private void buildHeap() {
  for ( int i = currentSize/2; i >= 0; i-- )
     percolateDown( i );
}
```

runtime:

Buildheap Analysis

n/4 nodes percolate at most 1 level
n/8 percolate at most 2 levels
n/16 percolate at most 3 levels
...



runtime:

The Math:

$$\sum_{i\geq 1} \frac{i}{2^i} = 2$$

$$\frac{n}{4} + \frac{2n}{8} + \frac{3n}{16} + \frac{4n}{32} + \cdots = \frac{n}{2}\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots\right] = \frac{n}{2}\sum_{i\geq 1}\frac{i}{2^i}$$

$$\begin{aligned} S = \sum_{i\geq 1}\frac{i}{2^i} &= \sum_{i\geq 1}\frac{1}{2^i} + \sum_{i\geq 1}\frac{i-1}{2^i} = 1 + \sum_{i\geq 1}\frac{i-1}{2^i} = 1 + \frac{1}{2}\sum_{i\geq 1}\frac{i-1}{2^{i-1}} \\ &= 1 + \frac{1}{2}\sum_{i\geq 0}\frac{i}{2^i} = 1 + \sum_{i\geq 1}\frac{i}{2^i} = 1 + \frac{S}{2} \end{aligned}$$

$$\begin{aligned} \sum_{i=1}^{\infty}\frac{i}{2^i} &= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots \\ &= (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots) + (\frac{1}{8} + \frac{1}{16} + \cdots) + \cdots \\ &= 1 + \frac{1}{2} + \frac{1}{4} + \cdots \\ &= 2 \end{aligned}$$

Heap sort

- Put n elements into a heap
- Perform n DeleteMins and put into an array

```
buildHeap();
for (int i = 0; i < n; i++)
        B[i] = DeleteMin();
```


Heap sort

```
// Sort array h of size n
void heapSort(int[]h, int n){
   for(int i = n/2; i >= 0; i--)
      percolateDown(h, i, n);
   for (int i = n-1; i > 0; i--){
      v = h[0];  h[0] = h[i]; h[i] = v;
      percolateDown(h, i, n);
   }
}
```

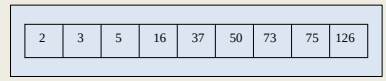
Analyzing code ("worst case")

Basic operations take "some amount of" constant time
- Arithmetic
- Assignment
- Access one Java field **or array index**
- Etc.

(This is an *approximation of reality*: a very useful "lie".)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of slower branch
Function Calls	Time of function's body
Recursion	Solve *recurrence equation*

Linear search



Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   for(int i=0; i < arr.length; ++i)
      if(arr[i] == k)
        return true;
   return false;
}
```

Best case: 6 steps = $O(1)$
Worst case: 5 * (arr.length)
= O(arr.length)

Analyzing Recursive Code

- Computing run-times gets interesting with recursion
- Say we want to perform some computation recursively on a list of size n
  - Conceptually, in each recursive call we:
    - Perform some amount of work, call it w(n)
    - Call the function recursively with a smaller portion of the list
- So, if we do w(n) work per step, and reduce the problem size in the next recursive call by 1, we do total work:

  T(n)=w(n)+T(n-1)
- With some base case, like T(1)=5=O(1)

Example Recursive code: sum array

Recursive:
- Recurrence is some constant amount of work O(1) done *n* times

```
int sum(int[] arr){
  return help(arr,0);
}
int help(int[]arr,int i) {
  if(i==arr.length)
    return 0;
  return arr[i] + help(arr,i+1);
}
```

Each time **help** is called, it does that O(1) amount of work, and then calls **help** again on a problem one less than previous problem size.

Recurrence Relation: $T(n) = O(1) + T(n-1)$

Solving Recurrence Relations

Say we have the following recurrence relation:

T(n)=6 + T(n-1)
T(1)=9 ←base case

Now we just need to solve it; that is, reduce it to a closed form.

Start by writing it out:

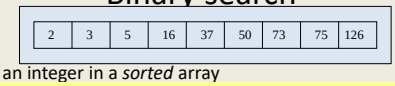
T(n)=6+T(n-1)
=6+6+T(n-2)
=6+6+6+T(n-3)
=6+6+6+…+6+T(1) = 6+6+6+…+6+9
=6k+T(n-k)
=6k+9, where k is the # of times we expanded T()

We expanded it out n-1 times, so

T(n)=6k+T(n-k)
=6(n-1)+T(1) = 6(n-1)+9
=6n+3 = O(n)

Or When does n-k=1?
Answer: when k=n-1

Binary search



Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
   int mid = (hi+lo)/2; //i.e., lo+(hi-lo)/2
   if(lo==hi)       return false;
   if(arr[mid]==k)  return true;
   if(arr[mid]< k)  return help(arr,k,mid+1,hi);
   else             return help(arr,k,lo,mid);
}
```

Binary search

Best case: 9 steps = $O(1)$

Worst case: $T(n) = 10 + T(n/2)$ where n is `hi-lo`

- $O(\log n)$ where n is `array.length`
- Solve *recurrence equation* to know that…

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
   return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
   int mid = (hi+lo)/2;
   if(lo==hi)       return false;
   if(arr[mid]==k)  return true;
   if(arr[mid]< k)  return help(arr,k,mid+1,hi);
   else             return help(arr,k,lo,mid);
}
```


Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$ $T(1) = 15$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case

sum array again

Two "obviously" linear algorithms: $T(n) = O(1) + T(n-1)$

Iterative:

```
int sum(int[] arr){
  int ans = 0;
  for(int i=0; i<arr.length; ++i)
     ans += arr[i];
  return ans;
}
```

Recursive:

- Recurrence is $c + c + \ldots + c$ for n times

```
int sum(int[] arr){
  return help(arr,0);
}
int help(int[]arr,int i) {
  if(i==arr.length)
     return 0;
  return arr[i] + help(arr,i+1);
}
```


What about a binary version of sum?

```
int sum(int[] arr){
   return help(arr,0,arr.length);
}

int help(int[] arr, int lo, int hi) {
   if(lo==hi)    return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```


What about a binary version of sum?

```
int sum(int[] arr){
   return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo==hi)    return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

Recurrence is $T(n) = O(1) + 2T(n/2)$

- $1 + 2 + 4 + 8 + \ldots$ for **log** n times
- $2^{(\log n)} - 1$ which is proportional to n (by definition of logarithm)

Easier explanation: it adds each number once while doing little else

"Obvious": You can't do better than $O(n)$ – have to read whole array

Parallelism teaser

- But suppose we could do two recursive calls *at the same time*
  - *Like having a friend do half the work for you!*

```
int sum(int[]arr){
   return help(arr,0,arr.length);
}
int help(int[]arr, int lo, int hi) {
   if(lo==hi)    return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

- If you have as many "friends of friends" as needed, the recurrence is now $T(n) = O(1) + 1T(n/2)$
  - $O(\log n)$: same recurrence as for `find`

Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

$T(n) = O(1) + T(n/2)$	logarithmic	$O(\log n)$
$T(n) = O(1) + 2T(n/2)$	linear	$O(n)$
$T(n) = O(1) + T(n-1)$	linear	$O(n)$
$T(n) = O(n) + T(n-1)$	quadratic	$O(n^2)$
$T(n) = O(1) + 2T(n-1)$	exponential	$O(2^n)$
$T(n) = O(n) + T(n/2)$	linear	$O(n)$
$T(n) = O(n) + 2T(n/2)$	loglinear	$O(n \log n)$