Today …

- Assignments
  - Assignment 6 due (don’t forget about the destructor!)
  - Assignment 7
- Binary Search Tree Deletion (cont.)
- Treesort [pp. 568-569]
- Heaps (if time) [Sect. 11.2]
Binary Search Tree Deletion

Removing (Deleting) Nodes

• Removing nodes is where things get tricky …

• If we could restrict remove to just leaf nodes it would be easy

• For general removal, we need to consider some special cases …
Removing Nodes

Cases to consider …

– The tree is empty!
– The item to be removed is not in the tree
– The node containing the item is a leaf node
– The node containing the item has one child
– The node containing the item has two children

Removing Nodes

• To remove a leaf node
  – We delete the node
  – And set the left or right pointer in the parent to NULL

• To remove a node with one child
  – The parent of the node to be deleted “adopts” the child
Removing Nodes

• **Hardest Case:** Remove a node with *two children*
  – Both children cannot be “adopted” by the parent

![Diagram showing a tree before and after removing node B]

Removing Nodes

• To remove a node with two children
  – one solution is to not delete the node …
  – and instead replace the item in the node with another node’s item

![Diagram showing a tree before and after replacing node B]
Removing Nodes

Which item do we then use as a replacement?

– There is more than one we can choose …
– Either the left-most node of the right subtree
– or the right-most node of the left subtree

Here, the right subtree does not have a left-most node

Removing Nodes

Which item do we then use as a replacement?

– There is more than one we can choose …
– Either the left-most node of the right subtree
– or the right-most node of the left subtree

A more typical example
Removing Nodes

• The left-most node of the right subtree is often referred to as the "inorder successor"
  – The node visited next in an inorder traversal

The inorder successor of B is C

Removing Nodes

• To find the inorder successor ...
  – Go to the right child
  – Then go left until you reach a node without a left child

Oops ... E has a right child!
Removing Nodes

- Remove the inorder successor …
  - It will either be a leaf or have one right child (why?)
  - Now we simply remove it …

Oops ... E has a right child!

Removing Nodes

Should we use **recursion** or **iteration** to find the inorder successor?

- Typically use **iteration**
- By looping until the left most node is found
- *Recall*: Navigating a path is easy using iteration (loops)
Exercise: Removing Nodes

• Form groups of 2
  – Your job is to write down the steps for remove
    
    void remove(const Key& key)
  – Remove should delete the first occurrence of the key
  – Only need to consider the case where the node to remove has two children
  – But consider both sub-cases: (a) when the inorder successor has a child, and (b) when it does not
**Binary Search Trees: Treesort**

Binary search trees can be used to sort a list of items

*Given a list of items …*

– Add them one by one into a Binary Search Tree
– Then do an *inorder* tree traversal (print nodes as visited)

• **Questions about this approach**

– Is this an efficient way to sort a list?
– What is the time complexity (worst case, best case)?
  
  • Worst case is $O(n^2)$
  
  • Best case is when tree is balanced … $O(n \log n)$
  
  • What does balanced mean for sorting?

---

### Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

What does this imply about the average case?
Heaps

- A heap is another ADT
- In the text …
  - Heaps are primarily introduced as a way to implement priority queues
  - Also used for heapsort
- We will primarily discuss heaps for heapsort
- But they are also a nice segue into balanced trees …
The Heap ADT

- A heap is very similar to a binary search tree
  - In fact, we are only going to talk about binary heaps ...
  - There are a number of variants however

- Unlike a binary search tree, a heap is always a complete tree!

- How does it do this? A min heap is just the opposite
  - By relaxing the ordering constraint
  - ... children of nodes have smaller values (max-heap)
  - thus, the root always has the max value

The Heap ADT

A (binary) “heap” is a complete binary tree such that:
- Every parent node has a larger search key value than its children (for max-heap)
- Trivially, an empty tree is a heap

![Heap Diagrams]

- This is a heap
- This is a not a heap ... why?
- This is a not a heap ... why?
The Heap ADT

The basic Heap ADT operations:

```cpp
// get the item with largest key in heap (alt findMin)
Entry findMax();

// delete the item with the largest key in the heap
void deleteMax();

// insert item into the heap
void insert(const Entry& newItem);
```

Array-Based Heap Implementation

Because a heap is always complete we can “efficiently” represent a heap using an array

- If node \( n \) is stored at index \( i \)
- The children of \( n \) are stored at \( 2i+1 \) (left) and \( 2i+2 \) (right)
- The parent of \( n \) is at position \( \lfloor (i-1)/2 \rfloor \)

![Binary Heap Example](image-url)
Array-Based Heap Implementation

• We assume that Heap has data members:
  
  Entry items[MAXITEMS]; // array of heap items
  int size; // number of items in the heap

Array-Based Heap Implementation

Entry Heap::findMax()
{
  if(!isEmpty())
    return items[0];
  else
    ... handle empty case ...
}

What is the worst case time complexity of findMax?
**Heap Implementation**

*Deleting* the max key

1. Remove root
2. Promote last item to root (becomes a “semiheap”)
3. “*Trickle down*” the new root
   a). If root’s search key smaller than largest child
   b). Then exchange (swap) them
   c). Trickle down new subtree root

---

**Heap Implementation**

Deleting the max key

```
Delete 10 (root)
```

```
9
8
5
2
4
```

```
4
9
8
5
2
```

```
4
9
8
5
2
```

```
9
8
5
2
```

```
9
8
5
2
```

```
9
8
5
2
```

```
9
8
5
2
```

```
9
8
5
2
```

Trickle down 4

Promote last item to root

**CPSC 223 – Fall 2010**
Array-Based Heap Implementation

void Heap::deleteMax()
{
    if(!isEmpty()) {
        size--;
        items[0] = items[size];
        heapRebuild(0);
    }
}

Array-Based Heap Implementation

void Heap::heapRebuild(int root)
{
    int child = 2*root + 1; // start with left child
    if(child < size) {
        int rightChild = child + 1;
        if(rightChild < size && items[rightChild] > items[child])
            child = rightChild; // set child to larger right child
        if(items[root] < items[child]) {
            swap(item[root], item[child]);
            heapRebuild(child);
        }
    }
}
Array-Based Heap Implementation

Cost of deletion

- How many comparisons and swaps?
- Worst case is when we trickle down to a leaf node
  - There are $O(\log n)$ such trickle-down steps … why?
- At each step we have
  - 2 (item) comparisons and 1 swap
  - Therefore, each trickle-down step is $O(1)$
- Deletion is worst case $O(\log n)$

Heap Implementation

**Inserting** a new item (… opposite of delete)

1. Add new item as leaf (maintain complete tree)
2. “**Trickle up**” the new root
   a). If item’s search key is greater than root’s search key
   b). Then exchange (swap) them
   c). Trickle up new root
Array-Based Heap Implementation

void Heap::insert(const Entry& newItem)
{
    if(size < MAXITEMS – 1) {
        int loc = size;
        int parent = (loc – 1)/2;
        items[loc] = newItem;
        while(parent >= 0 && items[parent] < newItem) {
            swap(items[parent], items[loc]);
            loc = parent;
            parent = (loc – 1)/2
        }
        size++;
    }
}

Array-Based Heap Implementation

Cost of insertion

• How many comparisons and swaps?

• The worst case is when we trickle up to the root
  – There are $O(\log n)$ such trickle-up steps … why?

• At each step we have
  – 1 (item) comparisons and 1 swap
  – Therefore, each trickle-up step is $O(1)$

• Insertion is worst case $O(\log n)$