CSCI 104
Priority Queues / Heaps

Mark Redekopp
David Kempe
Sandra Batista
Binary Tree Review

• Full binary tree: Binary tree, T, where
  – If height $h>0$ and both subtrees are full binary trees of height, $h-1$
  – If height $h==0$, then it is full by definition
  – (Tree where all leaves are at level $h$ and all other nodes have 2 children)

• Complete binary tree
  – Tree where levels 0 to $h-1$ are full and level $h$ is filled from left to right

• Balanced binary tree
  – Tree where subtrees from any node differ in height by at most 1
Array-based and Link-based

TREE IMPLEMENTATIONS
Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let’s say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = __________
  - Left_child(i) = __________
  - Right_child(i) = __________

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>em</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
parent(5) = _______
Left_child(5) = _______
Right_child(5) = _______
```
Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let’s say it starts at index 1 and index 0 is empty)

- Can you find the mathematical relation for finding node i's parent, left, and right child?
  - Parent(i) = \( \frac{i}{2} \)
  - Left_child(i) = \( 2*i \)
  - Right_child(i) = \( 2*i + 1 \)

```
Non-complete binary trees require much more bookeeping to store in arrays...usually link-based approaches are preferred
```
0-Based Indexing

• Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let’s say it starts at index 1 and index 0 is empty)

• Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  – Parent(i) = __________
  – Left_child(i) = _____________
  – Right_child(i) = _____________
0-Based Indexing

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let’s say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding the index of node i's parent, left, and right child?
  - Parent(i) = (i-1)/2
  - Left_child(i) = 2*i + 1
  - Right_child(i) = 2*i + 2

parent(5) = _______
Left_child(5) = _______
Right_child(5) = __________
D-ary Array-based Implementations

- Arrays can be used to store d-ary **complete** trees
  - Adjust the formulas derived for binary trees in previous slides in terms of d
Link-Based Approaches

- Much like a linked list but now with two pointers per Item
- Use NULL pointers to indicate no child
- Dynamically allocate and free items when you add/remove them

```cpp
#include<iostream>
using namespace std;

template <typename T>
struct BTItem {
    T val;
    BTItem<T>* left, right;
    BTItem<T>* parent;
};

// Bin. Search Tree
template <typename T>
class BinTree {
public:
    BinTree();
    ~BinTree();
    void add(const T& v);
    ...
private:
    BTItem<T>* root_;  // root of the binary tree
};
```

```cpp
class LinkedBST {
    BTItem<T>* root_;  // root of the linked binary tree
};
```
Link-Based Approaches

- Add(5)
- Add(6)
- Add(7)
Link-Based Approaches

• Add(5)
• Add(6)
• Add(7)

1. class LinkedBST:

   0x0 root_

2. 0x1c0 root_

   0x1c0 parent

   NULL

   Left

   NULL

   val

   5

   right

   NULL

3. 0x1c0 root_

   0x1c0 parent

   NULL

   Left

   NULL

   val

   5

   right

   0x2a0

4. 0x1c0 root_

   0x1c0 parent

   NULL

   Left

   NULL

   val

   5

   right

   0x2a0

   parent

   0x1c0

   Left

   NULL

   val

   6

   right

   NULL

   parent

   0x1c0

   Left

   NULL

   val

   6

   right

   0x0e0

   parent

   0x2a0

   Left

   NULL

   val

   7

   right

   NULL

   parent

   0x0e0
PRIORITy QUEUES
Traditional Queue

- **Traditional Queues**
  - Accesses/orders items based on POSITION (front/back)
  - Did not care about item's VALUE

- **Priority Queue**
  - Orders items based on VALUE
    - Either minimum or maximum
  - Items arrive in some arbitrary order
  - When removing an item, we always want the minimum or maximum depending on the implementation
    - Heaps that always yield the min value are called min-heaps
    - Heaps that always yield the max value are called max-heaps
  - Leads to a "sorted" list
  - Examples:
    - Think hospital ER, air-traffic control, etc.
Priority Queue

• What member functions does a Priority Queue have?
  – push(item) – Add an item to the appropriate location of the PQ
  – top() – Return the min./max. value
  – pop() - Remove the front (min. or max) item from the PQ
  – size() - Number of items in the PQ
  – empty() - Check if the PQ is empty
  – [Optional]: changePriority(item, new_priority)
    • Useful in many algorithms (especially AI and search algorithms)

• Implementations
  – Priority can be based upon intrinsic data-type being stored (i.e. operator<() of type T)
  – Priority can be passed in separately from data type, T,
    • Allows the same object to have different priorities based on the programmer's desire (i.e. same object can be assigned different priorities)
Priority Queue Efficiency

• If implemented as a sorted array list
  – Insert() = ______________
  – Top() = ______________
  – Pop() = ______________

• If implemented as an unsorted array list
  – Insert() = ______________
  – Top() = ______________
  – Pop() = ______________
Priority Queue Efficiency

• If implemented as a sorted array list
  – [Use back of array as location of top element]
  – Insert() = O(n)
  – Top() = O(1)
  – Pop() = O(1)

• If implemented as an unsorted array list
  – Insert() = O(1)
  – Top() = O(n)
  – Pop() = O(n)
HEAPS
Heap Data Structure

- Provides an efficient implementation for a priority queue
- Can think of heap as a complete binary tree that maintains the heap property:
  - **Heap Property**: Every parent is less-than (if min-heap) or greater-than (if max-heap) both children, but no ordering property between children
- Minimum/Maximum value is always the top element
Heap Operations

- **Push**: Add a new item to the heap and modify heap as necessary
- **Pop**: Remove min/max item and modify heap as necessary
- **Top**: Returns min/max
- **Since** heaps are complete binary trees we can use an array/vector as the container

```cpp
template <typename T>
class MinHeap
{
public:
    MinHeap(int init_capacity);
    ~MinHeap();
    void push(const T& item);
    T& top();
    void pop();
    int size() const;
    bool empty() const;
private:
    // Helper function
    void heapify(int idx);
    vector<T> items_; // or array
};
```
Array/Vector Storage for Heap

- Recall: Full binary tree (i.e. only the lowest-level contains empty locations and items added left to right) can be modeled as an array (let’s say it starts at index 1) where:
  - Parent(i) = i/2
  - Left_child(p) = 2*p
  - Right_child(p) = 2*p + 1
Array/Vector Storage for Heap

- We can also use 0-based indexing
  - Parent(i) = (i-1)/2
  - Left_child(p) = 2*p+1
  - Right_child(p) = 2*p + 2
Push Heap / TrickleUp

- Add item to first free location at bottom of tree
- Recursively promote it up while it is less than its parent
  - Remember valid heap all parents < children...so we need to promote it up until that property is satisfied

```cpp
void MinHeap<T>::push(const T& item) {
    items_.push_back(item);  // could be implemented recursively
    trickleUp(items_.size()-1);
}

void trickleUp(int loc) {
    int parent = loc/2;
    while(parent >= 1 && items_[loc] < items_[parent]) {
        swap(items_[parent], items_[loc]);
        loc = parent;
        parent = loc/2;
    }
}
```
Top()

• Top() simply needs to return first item

T& MinHeap<T>::top()
{
    if( empty() )
        throw(std::out_of_range());
    return items_[1];
}
Pop Heap / Heapify (TrickleDown)

- Pop utilizes the "heapify" algorithm (a.k.a. trickleDown)
- Takes last (greatest) node puts it in the top location and then recursively swaps it for the smallest child until it is in its right place

```cpp
void MinHeap<T>::pop()
{
    items_[1] = items_.back();
    items_.pop_back();
    heapify(1); // a.k.a. trickleDown()
}

void MinHeap<T>::heapify(int idx)
{
    if(idx == leaf node) return;
    int smallerChild = 2*idx; // start w/ left
    if(right child exists) {
        int rChild = smallerChild+1;
        if(items_[rChild] < items_[smallerChild])
            smallerChild = rChild;
    }
    if(items_[idx] > items_[smallerChild]){
        swap(items_[idx], items_[smallerChild]);
        heapify(smallerChild);
    }
}
```
Practice

Push(11)

Push(23)

Pop()

Pop()
Building a heap out of an array

HEAPSORT
Using a Heap to Sort

• If we could make a valid heap out of an arbitrary array, could we use that heap to sort our data?
  • Sure, just call top() and pop() \( n \) times
    – You'll get your data out in sorted order
  • How long would that take?
    – \( n \) calls to top() and pop()
      – top() = O(1)
      – pop = O(lg n)
  • Thus total time = O(n * lg n)

• But how long does it take to convert the array to a valid heap?
make_heap(): Converting An Unordered Array to a Heap

• We can convert an unordered array to a heap
  – std::make_heap does this
  – Let's see how...

• Basic operation: Given two heaps we can try to make one heap by unifying them with some new, arbitrary value but it likely won't be a heap

• How can we make a heap from this non-heap

• Heapify!! (we did this in pop() )
Converting An Array to a Heap

• To convert an array to a heap we can use the idea of first making heaps of both sub-trees and then combining the sub-trees (a.k.a. semi heaps) into one unified heap by calling heapify() on their parent()

• First consider all leaf nodes, are they valid heaps if you think of them as the root of a tree?
  – Yes!!

• So just start at the first non-leaf
Converting An Array to a Heap

• First consider all leaf nodes, are they valid heaps if you think of them as the root of a tree?
  – Yes!!
• So just start at the first non-leaf
  – Heapify(Loc. 4)

Leafs are valid heaps by definition
Converting An Array to a Heap

• Now that we have a valid heap, we can sort by top and popping...
• Can we do it in place?
  – Yes, Break the array into "heap" and "sorted" areas, iteratively adding to the "sorted" area

Swap top & last
heapify(1)

Swap top & last
heapify(1)
Converting An Array to a Heap

• Now that we have a valid heap, we can sort by top and popping…
• Can we do it in place?
  – Yes, Break the array into "heap" and "sorted" areas, iteratively adding to the "sorted" area

Swap top & last
heapify(1)

Swap top & last
heapify(1)
Converting An Array to a Heap

- Notice the result is in descending order.
- How could we make it ascending order?
  - Create a max heap rather than min heap.
Build-Heap Run-Time

• To build a heap from an arbitrary array require n calls to heapify.

• Heapify takes $O(\text{__________})$

• Let's be more specific:
  – Heapify takes $O(h)$
  – Because most of the heapify calls are made in the bottom of the tree (shallow $h$), it turns out heapify can be done in $O(n)$
C++ STL HEAP IMPLEMENTATION
STL Priority Queue

- Implements a heap
- Operations:
  - `push(new_item)`
  - `pop()`: removes but does not return top item
  - `top()`: return top item (item at back/end of the container)
  - `size()`
  - `empty()`
- By default, implements a **max** heap but can use comparator functors to create a **min**-heap
- Runtime: $O(\log(n))$ push and pop while all other functions are constant (i.e. $O(1)$)

```cpp
// priority_queue::push/pop
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
    priority_queue<int> mypq;
    mypq.push(30);
    mypq.push(100);
    mypq.push(25);
    mypq.push(40);
    cout << "Popping out elements...";
    while (!mypq.empty()) {
        cout << " " << mypq.top();
        mypq.pop();
    }
    cout << endl;
    return 0;
}
```

Code here will print

```
100 40 30 25
```
C++ less and greater

• If you're class already has operators < or > and you don't want to write your own functor you can use the C++ built-in functors: **less** and **greater**

• Less
  – Compares two objects of type T using the operator< defined for T

• Greater
  – Compares two objects of type T using the operator< defined for T

```cpp
template <typename T>
struct less
{
    bool operator()(const T& v1, const T& v2){
        return v1 < v2;
    }
};

template <typename T>
struct greater
{
    bool operator()(const T& v1, const T& v2){
        return v1 > v2;
    }
};
```
STL Priority Queue Template

- Template that allows type of element, container class, and comparison operation for ordering to be provided
- First template parameter should be type of element stored
- Second template parameter should be the container class you want to use to store the items (usually `vector<type_of_elem>`) 
- Third template parameters should be comparison functor that will define the order from first to last in the container

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main ()
{
    priority_queue<int, vector<int>, greater<int>> mypq;
    mypq.push(30); mypq.push(100); mypq.push(25);
    cout << "Popping out elements...";
    while (!mypq.empty()) {
        cout << " " << mypq.top();
        mypq.pop();
    }
    cout << endl;

    return 0;
}
```

Code here will print 25, 30, 100

greater<int> will yield a min-heap 
less<int> will yield a max-heap

Push(n): Mimics heap::push
Top(): Return last item
Pop(): Mimic heap::pop
STL Priority Queue Template

- For user defined classes, must implement operator<() for max-heap or operator>() for min-heap OR a custom functor
- Code here will pop in order:
  - Jane
  - Charlie
  - Bill

```cpp
// priority_queue::push/pop
#include <iostream>
#include <queue>
#include <string>
using namespace std;

class Item {
public:
  int score;
  string name;

  Item(int s, string n) { score = s; name = n;}
  bool operator>(const Item &lhs, const Item &rhs) const
  {
    if(lhs.score > rhs.score) return true;
    else return false;
  }
};

int main ()
{
  priority_queue<Item, vector<Item>, greater<Item> > mypq;
  Item i1(25,"Bill");      mypq.push(i1);
  Item i2(5,"Jane");      mypq.push(i2);
  Item i3(10,"Charlie");  mypq.push(i3);
  cout<< "Popping out elements...";
  while (!mypq.empty()) {
    cout<< " " << mypq.top().name;
    mypq.pop();
  }
}
```
More Details

• Behind the scenes `std::priority_queue` uses standalone functions defined in the `algorithm` library
  – push_heap
  – pop_heap
  – make_heap