TREES
Tree Definitions – Part 1

- **Definition**: A connected, acyclic (no cycles) graph with:
  - A root node, r, that has 0 or more subtrees
  - Exactly one path between any two nodes
- **In general**:
  - Nodes have exactly one parent (except for the root which has none) and 0 or more children
- **d-ary tree**
  - Tree where each node has at most d children
  - Binary tree = d-ary Tree with d=2

**Terms**:
- **Parent(i)**: Node directly above node i
- **Child(i)**: Node directly below node i
- **Siblings**: Children of the same parent
- **Root**: Only node with no parent
- **Leaf**: Node with 0 children
- **Height**: Number of nodes on longest path from root to any leaf
- **Subtree(n)**: Tree rooted at node n
- **Ancestor(n)**: Any node on the path from n to the root
- **Descendant(n)**: Any node in the subtree rooted at n
Tree Definitions – Part 2

• Tree height: maximum # of nodes on a path from root to any leaf

• **Full** $d$-ary tree, $T$, where
  – Every vertex has 0 or $d$ children and all leaf nodes are at the same level (i.e. adding 1 more node requires increasing the height of the tree)

• **Complete** $d$-ary tree
  – **Top $h-1$ levels are full** AND **bottom level is filled left-to-right**
  – Each level is filled left-to-right and a new level is not started until the previous one is complete

• **Balanced** $d$-ary tree
  – Tree where, for EVERY node, the subtrees for each child differ in height by at most 1

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
Tree Height

• A full or complete binary tree of \( n \) nodes has height,

\[ h = \lfloor \log_2(n + 1) \rfloor \]

  – This implies the minimum height of any tree with \( n \) nodes is

\[ \lceil \log_2(n + 1) \rceil \]

• The maximum height of a tree with \( n \) nodes is, ____

15 nodes => height \( \log_2(16) = 4 \)

5 nodes => height = __
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) = __________

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) = i/2
  - Left_child(i) = 2*i
  - Right_child(i) = 2*i + 1

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

• Now let's assume we start the root at index 0 of the array
• 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) = __________

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

• For an arbitrary (non-complete) d-ary tree we need to use pointer-based structures
  – 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 Item {
    T val;
    Item<T>* left, *right;
    Item<T>* parent;
};
// Bin. Search Tree
template <typename T>
class BinTree {
    public:
        BinTree();
        ~BinTree();
        void add(const T& v);
        ...
    private:
        Item<T>* root_;}
```

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
Link-Based Approaches

1. add(5)
2. add(6)
3. add(7)
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 graph and search algorithms)

- Priority can be based on...
  - Intrinsic data-type being stored (i.e. `operator<()` of type `T`)
  - Separate parameter from data type, `T`, and passed in which 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 *better-than* [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

![Min-Heap Diagram]

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
Heap Operations

• Push: Add a new item to the heap and modify heap as necessary
• Pop: Remove "best" (min/max) item and modify heap as necessary
• Top: Returns "best" item (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: A complete 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) = _______
  
  – Left_child(p) = _______
  
  – Right_child(p) = _______
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

```
void MinHeap<T>::push(const T& item)
{
  items_.push_back(item);
  trickleUp(items_.size()-1);
}

void MinHeap<T>::trickleUp(int loc)
{
  // could be implemented recursively
  int parent = __________;
  while(parent ______ &
        items_[loc] ___ items_[parent] )
  {
    swap(items_[parent], items_[loc]);
    loc = __________;
    parent = __________;
  }
}
```

Solutions at the end of these slides

---

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
\textbf{top()}

- \texttt{top()} simply needs to return first item

\begin{verbatim}
T const & MinHeap<T>::top()
{
    if( empty() )
        throw(std::out_of_range());
    return items_[1];
}
\end{verbatim}

\begin{tikzpicture}
    
ode[circle, fill=red!50] (root) at (0, 0) {1} child {node[circle] (s1) at (1, -1) {7} child {node[circle] (s2) at (2, -2) {18} child {node[circle] (s4) at (3, -3) {19} child {node[circle] (s8) at (4, -4) {28} child {node[circle] (s12) at (5, -5) {28}} child {node[circle] (s13) at (5, -5) {39}}}} child {node[circle] (s5) at (3, -3) {35}}}} child {node[circle] (s6) at (2, -2) {9} child {node[circle] (s11) at (3, -3) {14} child {node[circle] (s14) at (4, -4) {16}} child {node[circle] (s15) at (4, -4) {25}}}};
    
ode at (0, -2.5) {Top() returns 7};
\end{tikzpicture}
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()
}
```

```cpp
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);
    }
}
```
IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!

... WE'RE NOT INVITING YOU HOME NEXT YEAR.
Building a heap out of an arbitrary array

MAKE-HEAP / BUILD-HEAP
Motivation

• Suppose you are given an array of arbitrary data and you want to create a heap from that data
• You could
  – Allocate a second array for a heap,
  – Loop through the source array, and
  – Call \texttt{push(data[i])} on each iteration
  – Runtime: O(n*\log n)
• What if we said there was a way that:
  – Did not require a second array
  – Could build the heap in O(n)
**make_heap(): Converting An Unordered Array to a Heap**

- We will define a basic operation to convert the arbitrary array into a heap.

**Basic operation:** Given two smaller, valid heaps and one new value, merge/create a larger, valid heap.

**Approach:**
- Use the new value to "unify" the two smaller heaps by making it the root and the smaller heaps become subtrees.
- But this will likely violate the heap property.

**How can we make a heap from this non-heap?**
- **Heapify!!** (we did this in `pop()`)

Task: Merge / Create a new valid heap
Converting An Array to a Heap

- How can we use
- 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

- Call heapify() on each node in reverse order (from bottom to top)
- Optimization: Skip leaf nodes
  - If you consider all leaf nodes as individual heaps of size 1 (i.e. just that node as the root), they are already small, valid heaps
  - So just start at first non-leaf (i.e. heapify(3))

```cpp
void make_heap(vector<int>& dat) {
    for(int i=_______; i > ___; i-- ){
        // Heapify
    }
}
```
Make-Heap Run-Time

• To build a heap from an arbitrary array require \( n \) calls to heapify.
• For \text{\texttt{pop()}} we said heapify takes \( O(\,\,\,\,\,\,\,\,\,\,\,\,\,\,)\)
• Let's be more specific:
  – Heapify takes \( \Theta(h) \)
  – Because most of the heapify calls are made in the bottom of the tree (shallow \( h \)), it turns out heapify can be done in \( \Theta(\,\,\,)\)

\[
\begin{align*}
\text{void make_heap(vector<int>& dat) } & \text{ { } for(int i=dat.size()-1; i > 0; i--) } \\
& \text{ { } heapify(i); } \\
& \text{ } } \\
\end{align*}
\]

• \( n \) (all) calls do constant work (at \( h = 1 \))
• \( n/2 \) calls may have to do an extra swap (at \( h = 2 \))
• \( n/4 \) calls may have to do another swap (at \( h = 3 \))
• ... and only 1 call has \( h = \log n \)
• Totals: \( n + n/2 + n/4 + \ldots \)
• \( = n \left( 1 + \frac{1}{2} + \frac{1}{4} + 1/8 + \ldots \right) \)
• As \( h \) approaches infinity, the sum approaches \( 2n = \Theta(n) \)
Make-Heap Run-Time

- Or put another way, because most of the heapify calls are made in the bottom of the tree (shallow h), it turns out heapify can be done in $\theta(n)$

- Heapify takes $\theta(h)$
  - $n/2$ heapify calls with $h=1$  [i.e. the $n/2$ leaves]
  - $n/4$ calls with $h=2$
  - $n/8$ calls with $h=3$
  - Totals: $1*\frac{n}{2} + 2*\frac{n}{4} + 3*\frac{n}{8}$
  - $T(n)=\sum_{h=1}^{\log(n)} h * \frac{n}{2^{h}} = n * \sum_{h=1}^{\log(n)} h * \left(\frac{1}{2}\right)^{h}$
  - $T(n) = n * \theta(c) = \theta(n)$
Proving the Runtime of Make-Heap

• Let us prove that \( \sum_{h=1}^{\log(n)} h \times \left(\frac{1}{2}\right)^h \) is \( \Theta(1) \)

• \( T(n) = \sum_{h=1}^{\log(n)} h \times \left(\frac{1}{2}\right)^h < \sum_{h=1}^{\infty} h \times \left(\frac{1}{2}\right)^h \)

• Now recall: \( \sum_{h=1}^{\infty} (x)^h = \frac{1}{1-x} \) for \( x < 1 \)  \( [x=1/2 \text{ for this problem}] \)

• Now suppose we take the derivative of both sides

• \( \sum_{h=1}^{\infty} h \cdot (x)^{h-1} = \frac{1}{(1-x)^2} \)

• Suppose we multiply both sides by \( x \):
  \[ x \cdot \sum_{h=1}^{\infty} h \cdot (x)^{h-1} = \sum_{h=1}^{\infty} h \cdot (x)^h = \frac{x}{(1-x)^2} \]

• For \( x = \frac{1}{2} \) we have \( \sum_{h=1}^{\infty} h \cdot \left(\frac{1}{2}\right)^h = \frac{1}{2} \frac{1}{(1-\frac{1}{2})^2} = 2 \)

• Thus for Build-Heap: \( T(n) = n \times \sum_{h=1}^{\log(n)} h \times \left(\frac{1}{2}\right)^h = n \times \Theta(c) = \Theta(n) \)
Application of make-heap

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 to get data in sorted order
• How long would that take?
  – \( n \) calls to: top()=\( \Theta(1) \) and pop()=\( \Theta(\log n) \)
  – Thus total time = \( \Theta(n \cdot \log n) \)
• But how long does it take to convert the array to a valid heap?

© 2022 by Mark Redekopp. This content is protected and may not be shared, uploaded, or distributed.
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
• Notice the result is in descending order.
• How could we make it ascending order?
  – Create a max heap rather than min heap.
Reference/Optional

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
```
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
// priority_queue::push/pop
#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();
    }
}
```

Code here will print 25, 30, 100

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

Push(30)
```
0
30
```

Push(100)
```
0
1
30
100
```

Push(25)
```
0
1
2
25
100
30
```

Push(n): Mimics heap::push
Top(): Return last item
Pop(): Mimic heap::pop
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

- 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
#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 &rhs) const
    {
        if(this->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
    • https://en.cppreference.com/w/cpp/algorithm/push_heap
  – pop_heap
    • https://en.cppreference.com/w/cpp/algorithm/pop_heap
  – make_heap
    • https://en.cppreference.com/w/cpp/algorithm/make_heap
SOLUTIONS
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

```
void MinHeap<T>::push(const T& item)
{
    items_.push_back(item);
    trickleUp(items_.size()-1);
}

void MinHeap<T>::trickleUp(int loc)
{
    // could be implemented recursively
    int parent = loc/2;
    while(parent >= 1 &&
        items_[loc] < items_[parent] )
    {
        swap(items_[parent], items_[loc]);
        loc = parent;
        parent = loc/2;
    }
}
```

Solutions at the end of these slides