Heaps & Priority Queues in the C++ STL
2-3 Trees

CS 311 Data Structures and Algorithms
Lecture Slides
Friday, April 17, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

© 2005–2009 Glenn G. Chappell
Unit Overview
Tables & Priority Queues

Major Topics
✓ • Introduction to Tables
✓ • Priority Queues
✓ • Binary Heap algorithms
  • Heaps & Priority Queues in the C++ STL
  • 2-3 Trees
  • Other balanced search trees
✓ • Hash Tables
✓ • Prefix Trees
✓ • Tables in various languages
Review
Introduction to Tables

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array</th>
<th>Unsorted Array</th>
<th>Sorted Linked List</th>
<th>Unsorted Linked List</th>
<th>Binary Search Tree</th>
<th>Balanced (how?) Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve</td>
<td>Logarithmic</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Insert</td>
<td>Linear</td>
<td>Constant (?)</td>
<td>Linear</td>
<td>Constant</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>Delete</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Logarithmic</td>
</tr>
</tbody>
</table>

Idea #1: Restricted Table
- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced
- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: “Magic Functions”
- Use an unsorted array of key-data pairs. Allow array items to be marked as “empty”.
- Have a “magic function” that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:
- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables
A **Binary Heap** (usually just **Heap**) is a complete Binary Tree in which *no* node has a data item that is less than the data item in either of its children.

In practice, we often use “Heap” to refer to the array-based complete Binary Tree implementation.
Review
Binary Heap Algorithms [2/10]

We can use a Heap to implement a **Priority Queue**.

- Like a Table, but retrieve/delete only highest key.
  - Retrieve is called “getFront”.
- Key is called “priority”.
- Insert *any* key-data pair.

### Algorithms for the Three Primary Operations

- **GetFront**
  - Get the root node.
  - Constant time.
- **Insert**
  - Add new node to end of Heap, “trickle up”.
  - Logarithmic time if no reallocate-and-copy required.
    - Linear time if it may be required. Note: Heaps often do *not* manage their own memory, in which case the reallocation will not be part of the Heap operation.
- **Delete**
  - Swap first & last items, reduce size of Heap, “trickle down” root.
  - Logarithmic time.

Faster than linear time!
Review
Binary Heap Algorithms [3/10]

To insert into a Heap, add a new node at the end. Then “trickle up”.
  • If new value is greater than its parent, swap them. Repeat at new position.

![Binary Heap Insertion Diagram]
Review
Binary Heap Algorithms [4/10]

To delete the root item from a Heap, swap root and last items, and reduce size of Heap by one. Then “trickle down” the new root item.

- If the root is ≥ all its children, stop.
- Otherwise, swap the root item with its largest child and recursively fix the proper subtree.
Review
Binary Heap Algorithms [5/10]

Heap insert and delete are usually given a random-access range. The item to insert or delete is last item of the range; the rest is a Heap.

- Action of Heap insert:
  - That’s where we want to put the item, initially (right?).

- Action of Heap delete:
  - That’s where the swap puts it (right?).

Note that Heap algorithms can do all their work using swap.

- This usually allows for both speed and safety.
Review
Binary Heap Algorithms [6/10]

To turn a random-access range (array?) into a Heap, we could do $n-1$ Heap inserts.
- Each insert operation is $O(\log n)$, and so making a Heap in this way is $O(n \log n)$.

However, we can make a Heap faster than this.
- Place each item into a partially-made Heap, in backwards order.
- Trickle each item down through its descendants.
  - For most items, there are not very many of these.

Bottom items: no trickling necessary

This Heap “make” method is linear time!
Review
Binary Heap Algorithms [7/10]

Our last sorting algorithm is **Heap Sort**.

- This is a sort that uses Heap algorithms.
- We can think of it as using a Priority Queue, where the priority of an item is its value — except that the algorithm is in-place, using no separate data structure.
- Procedure: Make a Heap, then delete all items, using the delete procedure that places the deleted item in the top spot.
- We do a **make** operation, which is $O(n)$, and $n$ getFront/delete operations, each of which is $O(\log n)$.
- Total: $O(n \log n)$. 
Review
Binary Heap Algorithms [8/10]

Below: Heap make operation. Next slide: Heap deletion phase.

Start

Add 4

Add 1

Add 2

Add 3

Note: This is what happens in memory. This is just a picture of the logical structure.

Now the entire array is a Heap.
Review
Binary Heap Algorithms [9/10]

Heap deletion phase:

Start

Delete 4

Delete 3

Delete 2

Delete 1

Now the array is sorted.
Review
Binary Heap Algorithms [10/10]

Efficiency ☻
- Heap Sort is $O(n \log n)$.

Requirements on Data ☹
- Heap Sort requires random-access data.

Space Usage ☻
- Heap Sort is in-place.

Stability ☹
- Heap Sort is not stable.

Performance on Nearly Sorted Data ☹
- Heap Sort is not significantly faster or slower for nearly sorted data.

Notes
- Heap Sort can be generalized to handle sequences that are modified (in certain ways) in the middle of sorting.
- Recall that Heap Sort is used by Introsort, when the recursion depth of Quicksort exceeds the maximum allowed.
Heaps & Priority Queues in the C++ STL
Heap Algorithms

The C++ STL includes several Heap algorithms.

- These operate on ranges specified by pairs of random-access iterators.
  - Any random-access range can be a Heap: array, vector, deque, part of these, etc.
- An STL Heap is a Maxheap with an optional client-specified comparison.
- Heap algorithms are used by STL Priority Queues (std::priority_queue).

Example: \texttt{std::push_heap} (in \texttt{<algorithm>}) inserts into an existing Heap.

- Called as \texttt{std::push_heap(first, last)}.
- Assumes \texttt{[first, last)} is nonempty, and \texttt{[first, last-1)} is already a Heap.
- Inserts \texttt{*(last-1)} into the Heap.

Similarly:

- \texttt{std::pop_heap}
  - Heap delete operation. Puts the deleted element in \texttt{*(last-1)}.
- \texttt{std::make_heap}
  - Make a range into a Heap.
- \texttt{std::sort_heap}
  - Is given a Heap. Does a bunch of \texttt{pop_heap} calls.
  - Calling \texttt{make_heap} and then \texttt{sort_heap} does Heap Sort.
- \texttt{std::is_heap}
  - Tests whether a range is a Heap.
Heaps & Priority Queues in the C++ STL

`std::priority_queue` — Introduction

The STL has a Priority Queue: `std::priority_queue`, in `<queue>`.  
- STL documentation does not call `std::priority_queue` a “container”, but rather a “container adapter”.
- This is because `std::priority_queue` is explicitly a wrapper around some other container.

You get to pick what that container is.
- You say “`std::priority_queue<T, container<T> >`”.
  - “T” is the value type.
  - “`container`” can be `std::vector` or `std::deque`.
  - “`container<T>`” can be any standard-conforming `random-access` sequence container.
- `container` defaults to `std::vector`.
  - You can say just “`std::priority_queue<T>`” to get “`std::priority_queue<T, std::vector<T> >`”.

```
Heaps & Priority Queues in the C++ STL

`std::priority_queue` — Members

The member function names used by `std::priority_queue` are the same as those used by `std::stack`.

- Not those used by `std::queue`.
- Thus, `std::priority_queue` has "top", not "front".

Given a variable `pq` of type `std::priority_queue<T>`, you can do:

- `pq.top()`
- `pq.push(item)`
  - "item" is some value of type `T`.
- `pq.pop()`
- `pq.empty()`
- `pq.size()`
Heaps & Priority Queues in the C++ STL

std::priority_queue — Comparison

How do we specify an item’s priority?

- We really don’t need to know an item’s priority; we only need to know, given two items, which has the **higher** priority.
- Thus, we use a comparison, which defaults to `operator<`.
- A third, optional template parameter is a “comparison object”:

```
std::priority_queue<T, std::vector<T>, compare>
```

- Comparison objects work the same as those passed to STL sorting algorithms (`std::sort`, etc.) and STL Heap algorithms.
- So, for example, a priority queue of `ints` whose highest priority items are those with the lowest value, would have the following type:

```
std::priority_queue<int, std::vector<int>, std::greater<int>>()
```
Overview of Advanced Table Implementations

We will cover the following advanced Table implementations.

- Balanced Search Trees
  - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 children:
    - **2-3 Tree**
      - Up to 3 children
    - **2-3-4 Tree**
      - Up to 4 children
    - **Red-Black Tree**
      - Binary-tree representation of a 2-3-4 tree
    - Or back up and try a balanced Binary Tree again:
      - **AVL Tree**

- Alternatively, forget about trees entirely:
  - **Hash Tables**

- Finally, “the Radix Sort of Table implementations”:
  - **Prefix Tree**
2-3 Trees
Introduction & Definition [1/3]

Obviously (?) a Binary Search Tree is a useful idea. The problem is keeping it balanced.
- Or at least keeping the height small.

It turns out that small height is much easier to maintain if we allow a node to have more than 2 children.

But if we do this, how do we maintain the “search tree” concept?
- We generalize the idea of an inorder traversal.
- For each pair of consecutive subtrees, a node has one data item lying between the values in these subtrees.
A Binary-Search-Tree style node is a **2-node**.
- This is a node with 2 subtrees and 1 data item.
- The item’s value lies between the values in the two subtrees.

In a “2-3 Tree” we also allow a node to be a **3-node**.
- This is a node with 3 subtrees and 2 data items.
- Each of the 2 data items has a value that lies between the values in the corresponding pair of consecutive subtrees.

Later, we will look at “2-3-4 trees”, which can also have **4-nodes**.
A **2-3 Search Tree** (generally we just say **2-3 Tree**) is a tree with the following properties.

- All nodes contain either 1 or 2 data items.
  - If 2 data items, then the first is ≤ the second.
- All leaves are at the same level.
- All non-leaves are either 2-*nodes* or 3-*nodes*.
  - They must have the associated order properties.
2-3 Trees
Operations — Traverse & Retrieve

How do we **traverse** a 2-3 Tree?
- We generalize the procedure for doing an **inorder traversal** of a Binary Search Tree.
  - For each leaf, go through the items in it.
  - For each non-leaf 2-node:
    - Traverse subtree 1.
    - Do item.
    - Traverse subtree 2.
  - For each non-leaf 3-node:
    - Traverse subtree 1.
    - Do item 1.
    - Traverse subtree 2.
    - Do item 2.
    - Traverse subtree 3.
- This procedure lists all the items in sorted order.

How do we **retrieve** by key in a 2-3 Tree?
- Start at the root and proceed downward, making comparisons, just as in a Binary Search Tree.
- 3-nodes make the procedure slightly more complex.
2-3 Trees
Operations — Insert & Delete

How do we **insert** & **delete** in a 2-3 Tree?

- These are the tough problems.
- It turns out that both have efficient \([O(\log n)]\) algorithms, which is why we like 2-3 Trees.
Basic ideas behind the 2-3 Tree insert algorithm:

- Allow nodes to expand when legal.
- If a node gets too big (3 items), split the subtree rooted at that node and propagate the middle item upward.
- If we end up splitting the entire tree, then we create a new root node, and all the leaves advance one level simultaneously.

Example 1: Insert 10.
2-3 Trees
Operations — Insert [2/4]

Example 2: Insert 5.
- Over-full nodes are blue.
Example 3: Insert 5.
- Here we see how a 2-3 Tree increases in height.
2-3 Trees
Operations — Insert [4/4]

In the middle of a 2-3 Tree insertion, overfull nodes are always leaves or 4-nodes.

• (Of course 4-nodes are illegal.)
• This is why we can always split at the middle item.

2-3 Tree insertion can be thought of recursively.

• Insert into a node.
  ▪ This node will be a leaf the first time.

• If the node is overfull, split and move the middle item up.
  ▪ We split the subtree rooted at the node. If the node is a leaf, this is easy. If the node is not a leaf, this is more work, but not hard to understand.

• Moving an item up is inserting into the parent.
  ▪ So we recurse.
  ▪ Or we make increase the height (by making a new root), and we are done.
2-3 Trees
TO BE CONTINUED ...