Binary Trees, Binary Search Trees

www.cs.ust.hk/~huamin/COMP171/bst.ppt
Trees

• Linear access time of linked lists is prohibitive
  – Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is $O(\log N)$?
Trees

- A tree is a collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node $r$ (the root), and zero or more nonempty subtrees

Figure 4.1 Generic tree
Some Terminologies

- **Child** and **parent**
  - Every node except the root has one parent
  - A node can have an arbitrary number of children
- **Leaves**
  - Nodes with no children
- **Sibling**
  - Nodes with same parent
Some Terminologies

- **Path**
- **Length**
  - number of edges on the path
- **Depth of a node**
  - length of the unique path from the root to that node
  - The depth of a tree is equal to the depth of the deepest leaf
- **Height of a node**
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
  - The height of a tree is equal to the height of the root
- **Ancestor and descendant**
  - Proper ancestor and proper descendant
Example: UNIX Directory

Figure 4.5 UNIX directory
Binary Trees

- A tree in which no node can have more than two children.

- The depth of an “average” binary tree is considerably smaller than $N$, even though in the worst case, the depth can be as large as $N - 1$. 
Example: Expression Trees

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

Figure 4.14 Expression tree for \((a + b \times c) + ((d \times e + f) \times g)\)
Tree traversal

• Used to print out the data in a tree in a certain order

• Pre-order traversal
  – Print the data at the root
  – Recursively print out all data in the left subtree
  – Recursively print out all data in the right subtree
Preorder, Postorder and Inorder

- Preorder traversal
  - node, left, right
  - prefix expression

Figure 4.14 Expression tree for \((a + b \times c) + ((d \times e + f) \times g)\)
Preorder, Postorder and Inorder

• Postorder traversal
  – left, right, node
  – postfix expression
    • abc*+de*f+g*

• Inorder traversal
  – left, node, right
  – infix expression
    • a+b*c+d*e+f*g

Figure 4.14 Expression tree for \((a + b * c) + ((d * e + f) * g)\)
• Preorder

```
/usr
  mark
    book
      ch1.r
      ch2.r
      ch3.r
  course
    cop3530
      fall98
        syl.r
      spr99
        syl.r
      sum99
        syl.r
    junk
  alex
    junk
  bill
    work
  course
    cop3212
      fall98
        grades
        prog1.r
        prog2.r
      fall99
        prog2.r
        prog1.r
        grades
```
Preorder, Postorder and Inorder

Algorithm  Preorder($x$)
Input: $x$ is the root of a subtree.
1.  if $x \neq$ NULL
2.      then output key($x$);
3.  Preorder(left($x$));
4.  Preorder(right($x$));

Algorithm  Postorder($x$)
Input: $x$ is the root of a subtree.
1.  if $x \neq$ NULL
2.      then Postorder(left($x$));
3.  Postorder(right($x$));
4.  output key($x$);

Algorithm  Inorder($x$)
Input: $x$ is the root of a subtree.
1.  if $x \neq$ NULL
2.      then Inorder(left($x$));
3.  output key($x$);
4.  Inorder(right($x$));
Binary Trees

- Possible operations on the Binary Tree ADT
  - parent
  - left_child, right_child
  - sibling
  - root, etc

- Implementation
  - Because a binary tree has at most two children, we can keep direct pointers to them:
    ```c
    struct BinaryNode
    {
        Object element;     // The data in the node
        BinaryNode *left;   // Left child
        BinaryNode *right;  // Right child
    };
    ```
compare: Implementation of a general tree

Figure 4.2 A tree

Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2
Binary Search Trees

• Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

Binary search tree property

– For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X
Binary Search Trees

A binary search tree

Not a binary search tree
Binary search trees

Two binary search trees representing the same set:

- Average depth of a node is $O(\log N)$; maximum depth of a node is $O(N)$
Implementation

template <class Comparable>
class BinarySearchTree;

template <class Comparable>
class BinaryNode
{
    Comparable element;
    BinaryNode *left;
    BinaryNode *right;

    BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt ) :
    element( theElement ), left( lt ), right( rt ) {}
friend class BinarySearchTree<Comparable>;
};

Figure 4.16 The BinaryNode class
Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.

```
root

  15

  < 15

  > 15
```
Example: Search for 9 ...

Search for 9:

1. compare 9:15 (the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!
Searching (Find)

- Find X: return a pointer to the node that has key X, or NULL if there is no such node

```cpp
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
find( const Comparable & x, BinaryNode<Comparable> * t ) const
{
    if( t == NULL )
        return NULL;
    else if( x < t->element )
        return find( x, t->left );
    else if( t->element < x )
        return find( x, t->right );
    else
        return t;    // Match
}
```

- Time complexity
  - O(height of the tree)
Inorder traversal of BST

• Print out all the keys in sorted order

Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
findMin/ findMax

- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```cpp
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

- Similarly for findMax
- Time complexity = O(height of the tree)
insert

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed

- Time complexity = $O(\text{height of the tree})$
delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.
Three cases:

(1) the node is a leaf
   - Delete it immediately

(2) the node has one child

Figure 4.24 Deletion of a node (4) with one child, before and after
delete

(3) the node has 2 children
   - replace the key of that node with the minimum element at the right subtree
   - delete the minimum element

   • Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

![Figure 4.25 Deletion of a node (2) with two children, before and after](image)

• Time complexity = $O(\text{height of the tree})$
Priority Queues – Binary Heaps

homes.cs.washington.edu/~anderson/iucee/Slides_326.../Heaps.ppt
Recall Queues

• FIFO: First-In, First-Out

• Some contexts where this seems right?

• Some contexts where some things should be allowed to skip ahead in the line?
Queues that Allow Line Jumping

• Need a new ADT

• Operations: Insert an Item, Remove the “Best” Item

```
6  15  12  45
2  23  18  3
18 7
```

`insert` `deleteMin`
Priority Queue ADT

1. PQueue data: collection of data with priority

2. PQueue operations
   - insert
   - deleteMin

3. PQueue property: for two elements in the queue, $x$ and $y$, if $x$ has a lower priority value than $y$, $x$ will be deleted before $y$
Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

- Anything greedy
## Potential Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>O(n)</td>
<td>O(1)*</td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Recall From Lists, Queues, Stacks

• Use an ADT that corresponds to your needs

• The right ADT is efficient, while an overly general ADT provides functionality you aren’t using, but are paying for anyways

• Heaps provide $O(\log n)$ worst case for both insert and deleteMin, $O(1)$ average insert
Binary Heap Properties

1. Structure Property
2. Ordering Property
Tree Review

- **root(T):**
- **leaves(T):**
- **children(B):**
- **parent(H):**
- **siblings(E):**
- **ancestors(F):**
- **descendents(G):**
- **subtree(C):**
More Tree Terminology

(depth(B):

(height(G):

(degree(B):

(branching factor(T):
Brief interlude: Some Definitions:

A **Perfect** binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

- **Height** $h$
- $2^{h+1} - 1$ nodes
- $2^h - 1$ non-leaves
- $2^h$ leaves
A binary heap is a **complete** binary tree. **Complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:
Representing Complete Binary Trees in an Array

From node $i$:
left child:
right child:
parent:

implicit (array) implementation:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Why this approach to storage?
Heap Order Property

Heap order property: For every non-root node $X$, the value in the parent of $X$ is less than (or equal to) the value in $X$.

not a heap
Heap Operations

• **findMin:**
• **insert(val):** percolate up.
• **deleteMin:** percolate down.
Heap – Insert(val)

Basic Idea:
1. Put val at “next” leaf position
2. Percolate up by repeatedly exchanging node until no longer needed
Insert: percolate up

```
      10
     /  \
    20   80
   /  \   /  \
  40   60 85   99
 /  \ /  \ /  \ /  \ /
50  700 65 15 80 15 85 99
```

```
void insert(Object o) {
    assert(!isFull());
    size++;
    newPos =
        percolateUp(size, o);
    Heap[newPos] = o;
}

int percolateUp(int hole, Object val) {
    while (hole > 1 &&
           val < Heap[hole/2])
        Heap[hole] = Heap[hole/2];
    hole /= 2;
    return hole;
}

 runtime:

(Code in book)
Heap – Deletemin

Basic Idea:
1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.
DeleteMin: percolate down
DeleteMin Code (Optimized)

Object deleteMin() {
    assert(!isEmpty());
    returnVal = Heap[1];
    size--;
    newPos =
        percolateDown(1,
            Heap[size+1]);
    Heap[newPos] =
        Heap[size + 1];
    return returnVal;
}

int percolateDown(int hole, Object val) {
    while (2*hole <= size) {
        left = 2*hole;
        right = left + 1;
        if (right <= size &&
            Heap[right] < Heap[left])
            target = right;
        else
            target = left;

        if (Heap[target] < val) {
            Heap[hole] = Heap[target];
            hole = target;
        }
        else
            break;
    }
    return hole;
}
Insert: 16, 32, 4, 69, 105, 43, 2
Data Structures

Binary Heaps
Building a Heap

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

• Adding the items one at a time is $O(n \log n)$ in the worst case

• I promised $O(n)$ for today
Working on Heaps

• What are the two properties of a heap?
  – Structure Property
  – Order Property

• How do we work on heaps?
  – Fix the structure
  – Fix the order
Add elements arbitrarily to form a complete tree. Pretend it’s a heap and fix the heap-order property!
Buildheap pseudocode

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

runtime:
BuildHeap: Floyd’s Method
BuildHeap: Floyd’s Method

12
  /   \
 5    11
 /      |
3       2
 / \\   / \
4  8  1 7 6
   /   \
 9
BuildHeap: Floyd’s Method
BuildHeap: Floyd’s Method

```plaintext
12

5

3 10

4 8 1 7

6

11

2 9

12

5

3

4 8

10 7

6

11

2

9

12

5

3

1

6

9

11

4 8 10 7 11
```
BuildHeap: Floyd’s Method
Finally…

runtime:
More Priority Queue Operations

• **decreaseKey**
  – given a pointer to an object in the queue, reduce its priority value

  Solution: change priority and

  ________________________________

• **increaseKey**
  – given a pointer to an object in the queue, increase its priority value

  Why do we need a *pointer*? Why not simply data value?
  Solution: change priority and

  ________________________________
More Priority Queue Operations

• **Remove**(objPtr)
  – given a pointer to an object in the queue, remove the object from the queue

  **Solution**: set priority to negative infinity, percolate up to root and deleteMin

• **FindMax**
Facts about Heaps

Observations:
• Finding a child/parent index is a multiply/divide by two
• Operations jump widely through the heap
• Each percolate step looks at only two new nodes
• Inserts are at least as common as deleteMins

Realities:
• Division/multiplication by powers of two are equally fast
• Looking at only two new pieces of data: bad for cache!
• With huge data sets, disk accesses dominate
Cycles to access:

- CPU
- Cache
- Memory
- Disk
A Solution: $d$-Heaps

- Each node has $d$ children
- Still representable by array
- Good choices for $d$:
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block
Tries

hwww.mathcs.emory.edu/~cheung/Fcourses/323/Syllabus/book/PowerPoint/tries.ppt
Preprocessing Strings

• Preprocessing the pattern speeds up pattern matching queries
  – After preprocessing the pattern, KMP’s algorithm performs pattern matching in time proportional to the text size

• If the text is large, immutable and searched for often (e.g., works by Shakespeare), we may want to preprocess the text instead of the pattern

• A trie is a compact data structure for representing a set of strings, such as all the words in a text
  – A trie supports pattern matching queries in time proportional to the pattern size
Standard Tries

- The standard trie for a set of strings $S$ is an ordered tree such that:
  - Each node but the root is labeled with a character
  - The children of a node are alphabetically ordered
  - The paths from the external nodes to the root yield the strings of $S$

- Example: standard trie for the set of strings $S = \{ \text{bear, bell, bid, bull, buy, sell, stock, stop} \}$

![Trie Diagram]

Tries
Analysis of Standard Tries

• A standard trie uses $O(n)$ space and supports searches, insertions and deletions in time $O(dm)$, where:
  
  - $n$ total size of the strings in S
  - $m$ size of the string parameter of the operation
  - $d$ size of the alphabet
Word Matching with a Trie

- Insert the words of the text into trie.
- Each leaf is associated with one particular word.
- Leaf stores indices where associated word begins ("see" starts at index 0 & 24, leaf for "see" stores those indices.)
Compressed Tries

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of “redundant” nodes
- Ex. the “i” and “d” in “bid” are “redundant” because they signify the same word
Compact Representation

- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Uses $O(s)$ space, where $s$ is the number of strings in the array
  - Serves as an auxiliary index structure

```
S[0] = see
S[1] = bear
S[2] = sell
S[3] = stock
S[4] = bull
S[5] = buy
S[7] = hear
S[8] = bell
S[9] = stop
```
Suffix Trie

- The suffix trie of a string $X$ is the compressed trie of all the suffixes of $X$
Analysis of Suffix Tries

- Compact representation of the suffix trie for a string $X$ of size $n$ from an alphabet of size $d$
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in $X$ in $O(dm)$ time, where $m$ is the size of the pattern
  - Can be constructed in $O(n)$ time
Encoding Trie (1)

• A code is a mapping of each character of an alphabet to a binary code-word

• A prefix code is a binary code such that no code-word is the prefix of another code-word

• An encoding trie represents a prefix code
  – Each leaf stores a character
  – The code word of a character is given by the path from the root to the leaf storing the character (0 for a left child and 1 for a right child)
Encoding Trie (2)

- Given a text string $X$, we want to find a prefix code for the characters of $X$ that yields a small encoding for $X$
  - Frequent characters should have short code-words
  - Rare characters should have long code-words
- Example
  - $X = \text{abracadabra}$
  - $T_1$ encodes $X$ into 29 bits
  - $T_2$ encodes $X$ into 24 bits
The End
Huffman’s Algorithm

- Given a string $X$, Huffman’s algorithm construct a prefix code that minimizes the size of the encoding of $X$.
- It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$.
- A heap-based priority queue is used as an auxiliary structure.

Algorithm $HuffmanEncoding(X)$

Input string $X$ of size $n$

Output optimal encoding trie for $X$

1. $C \leftarrow distinctCharacters(X)$
2. $computeFrequencies(C, X)$
3. $Q \leftarrow$ new empty heap
4. for all $c \in C$
   1. $T \leftarrow$ new single-node tree storing $c$
   2. $Q.insert(getFrequency(c), T)$
5. while $Q.size() > 1$
   1. $f_1 \leftarrow Q.min()$
   2. $T_1 \leftarrow Q.removeMin()$
   3. $f_2 \leftarrow Q.min()$
   4. $T_2 \leftarrow Q.removeMin()$
   5. $T \leftarrow join(T_1, T_2)$
   6. $Q.insert(f_1 + f_2, T)$
6. return $Q.removeMin()$
Example

$X = \text{abracadabra}$

Frequencies

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Diagram:

- Tries tree with frequencies:
  - a: 5
  - b: 2
  - c: 1
  - d: 1
  - r: 2

- Example tree:
  - a: 6
  - b: 2
  - c: 4

Diagram arrows indicate transformations between trees.
One More Operation

• Merge two heaps

• Add the items from one into another?
  – $O(n \log n)$

• Start over and build it from scratch?
  – $O(n)$
CSE 326: Data Structures

Priority Queues
Leftist Heaps & Skew Heaps
New Heap Operation: Merge

Given two heaps, merge them into one heap

– first attempt: insert each element of the smaller heap into the larger.

  \textit{runtime:}

– second attempt: concatenate binary heaps’ arrays and run \texttt{buildHeap}.

  \textit{runtime:}
Leftist Heaps

Idea:
Focus all heap maintenance work in one small part of the heap

Leftist heaps:
1. Most nodes are on the left
2. All the merging work is done on the right
Definition: Null Path Length

null path length \((npl)\) of a node \(x\) = the number of nodes between \(x\) and a null in its subtree

OR

\(npl(x) = \min\) distance to a descendant with 0 or 1 children

- \(npl(\text{null}) = -1\)
- \(npl(\text{leaf}) = 0\)
- \(npl(\text{single-child node}) = 0\)

Equivalent definitions:

1. \(npl(x)\) is the height of largest complete subtree rooted at \(x\)
2. \(npl(x) = 1 + \min\{npl(\text{left}(x)), npl(\text{right}(x))\}\)
Leftist Heap Properties

• Heap-order property
  – parent’s priority value is ≤ to childrens’ priority values
  – result: minimum element is at the root

• Leftist property
  – For every node x, npl(left(x)) ≥ npl(right(x))
  – result: tree is at least as “heavy” on the left as the right
Are These Leftist?

Every subtree of a leftist tree is leftist!
Right Path in a Leftist Tree is Short (#1)

**Claim:** The right path is as short as *any* in the tree.

**Proof:** (By contradiction)

Pick a shorter path: \( D_1 < D_2 \)

Say it diverges from right path at \( x \)

\[ npl(L) \leq D_1 - 1 \quad \text{because of the path of length } D_1 - 1 \text{ to null} \]

\[ npl(R) \geq D_2 - 1 \quad \text{because every node on right path is leftist} \]

Leftist property at \( x \) violated!
Right Path in a Leftist Tree is Short (#2)

Claim: If the right path has $r$ nodes, then the tree has at least $2^r - 1$ nodes.

Proof: (By induction)

Base case: $r=1$. Tree has at least $2^1 - 1 = 1$ node

Inductive step: assume true for $r' < r$. Prove for tree with right path at least $r$.

1. Right subtree: right path of $r-1$ nodes
   \[\Rightarrow 2^{r-1}-1\] right subtree nodes (by induction)
2. Left subtree: also right path of length at least $r-1$ (by previous slide)
   \[\Rightarrow 2^{r-1}-1\] left subtree nodes (by induction)

Total tree size: $(2^{r-1}-1) + (2^{r-1}-1) + 1 = 2^r - 1$
Why do we have the leftist property?

Because it guarantees that:

• the *right path is really short* compared to the number of nodes in the tree

• A leftist tree of N nodes, has a right path of at most \( \lg (N+1) \) nodes

**Idea** – perform all work on the right path
Merge two heaps (basic idea)

• Put the smaller root as the new root,
• Hang its left subtree on the left.
• Recursively merge its right subtree and the other tree.
Merging Two Leftist Heaps

- \( \text{merge}(T_1, T_2) \) returns one leftist heap containing all elements of the two (distinct) leftist heaps \( T_1 \) and \( T_2 \)
Merge Continued

If $npl(R') > npl(L_1)$

$R' = \text{Merge}(R_1, T_2)$

runtime:
Let’s do an example, but first…

Other Heap Operations

• insert ?

• deleteMin ?
Operations on Leftist Heaps

- **merge** with two trees of total size $n$: $O(\log n)$
- **insert** with heap size $n$: $O(\log n)$
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node heap

\[ \triangle \quad \circ \quad \overset{\text{merge}}{\longrightarrow} \triangle \]

- **deleteMin** with heap size $n$: $O(\log n)$
  - remove and return root
  - merge left and right subtrees
Leftest Merge Example

(special case)
Sewing Up the Example

Done?
Finally…
Leftist Heaps: Summary

Good

•

•

Bad

•

•
Random Definition: Amortized Time

am·or·tized time:
Running time limit resulting from “writing off” expensive runs of an algorithm over multiple cheap runs of the algorithm, usually resulting in a lower overall running time than indicated by the worst possible case.

If M operations take total $O(M \log N)$ time, amortized time per operation is $O(\log N)$

Difference from average time:
Skew Heaps

Problems with leftist heaps
- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is “often” heavy and requires a switch

Solution: skew heaps
- “blindly” adjusting version of leftist heaps
- merge always switches children when fixing right path
- amortized time for: merge, insert, deleteMin = O(log n)
- however, worst case time for all three = O(n)
Merging Two **Skew** Heaps

Only one step per iteration, with children *always* switched
Example

merge

merge

merge
void merge(heap1, heap2) {
    case {
        heap1 == NULL: return heap2;
        heap2 == NULL: return heap1;
        heap1.findMin() < heap2.findMin() :
            temp = heap1.right;
            heap1.right = heap1.left;
            heap1.left = merge(heap2, temp);
            return heap1;
        otherwise:
            return merge(heap2, heap1);
    }
}
Runtime Analysis: Worst-case and Amortized

- No worst case guarantee on right path length!
- All operations rely on merge

⇒ worst case complexity of all ops =

- Probably won’t get to amortized analysis in this course, but see Chapter 11 if curious.
- Result: $M$ merges take time $M \log n$

⇒ amortized complexity of all ops =
Comparing Heaps

• Binary Heaps
  • Leftist Heaps

• d-Heaps
  • Skew Heaps

Still room for improvement! (Where?)
Data Structures
Binominal Queues
Yet Another Data Structure: Binomial Queues

• Structural property
  – Forest of binomial trees with at most one tree of any height

• Order property
  – Each binomial tree has the heap-order property

What’s a forest?
What’s a binomial tree?
The Binomial Tree, $B_h$

- $B_h$ has height $h$ and exactly $2^h$ nodes
- $B_h$ is formed by making $B_{h-1}$ a child of another $B_{h-1}$
- Root has exactly $h$ children
- Number of nodes at depth $d$ is binomial coeff.
  - Hence the name; we will not use this last property

\[
\binom{h}{d}
\]
Binomial Queue with $n$ elements

Binomial Q with $n$ elements has a unique structural representation in terms of binomial trees!

Write $n$ in binary:

$n = 1101 \text{ (base 2)} = 13 \text{ (base 10)}$
Properties of Binomial Queue

• At most one binomial tree of any height

• $n$ nodes $\Rightarrow$ binary representation is of size $\ell$?
  $\Rightarrow$ deepest tree has height $\ell$?
  $\Rightarrow$ number of trees is $\ell$?

Define: $\text{height}(\text{forest } F) = \max_{\text{tree } T \text{ in } F} \{ \text{height}(T) \}$

Binomial Q with $n$ nodes has height $\Theta(\log n)$
Operations on Binomial Queue

• Will again define *merge* as the base operation
  – insert, deleteMin, buildBinomialQ will use merge

• Can we do increaseKey efficiently? decreaseKey?

• What about findMin?
Merging Two Binomial Queues

Essentially like adding two binary numbers!

1. Combine the two forests
2. For \( k \) from 0 to maxheight {
   a. \( m \leftarrow \) total number of \( B_k \)'s in the two BQs
   b. if \( m=0 \): continue;
   c. if \( m=1 \): continue;
   d. if \( m=2 \): combine the two \( B_k \)'s to form \( B_{k+1} \)
   e. if \( m=3 \): retain one \( B_k \) and combine the other two to form a \( B_{k+1} \)
}

Claim: When this process ends, the forest has at most one tree of any height
Example: Binomial Queue Merge

H1:

H2:
Example: Binomial Queue Merge

H1: 

H2: 

-1

1

7

2

1

3

8

11

5

6

3

5

21

9

6

7
Example: Binomial Queue Merge

H1:

H2:
Example: Binomial Queue Merge

H1:

H2:
Example: Binomial Queue Merge

H1: 

\[-1 \quad 2 \quad 1 \quad 3 \quad 8 \quad 11 \quad 5 \quad 1 \quad 7 \quad 3 \quad 5 \quad 6 \quad 21 \quad 9 \quad 6 \quad 7\]

H2:
Example: Binomial Queue

Merge

H1:  H2:

```
-1
 /    \
2 1 3
 /     /
8 11 5
```

```
1
 /    /
7 3
 /    /
5 21
 /    /
6 9 6
 /    /
7
```
Complexity of Merge

Constant time for each height
Max number of heights is: \( \log n \)

\[ \Rightarrow \quad \text{worst case running time} = \Theta( ??? ) \]
Insert in a Binomial Queue

Insert($x$): Similar to leftist or skew heap

Runtime
Worst case complexity: same as merge
$O(\text{____})$

Average case complexity: $O(1)$
Why?? Hint: Think of adding 1 to 1101
deleteMin in Binomial Queue
Similar to leftist and skew heaps….
deleteMin: Example

BQ

find and delete smallest root

merge BQ (without the shaded part) and BQ’
deleteMin: Example

Result:

runtime: