

# Priority Queues and Huffman Trees

## the Heap

storing the heap with a vector  
deleting from the heap

## Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

## Huffman Trees

encoding messages  
a recursive tree creation algorithm

### 1 the Heap

storing the heap with a vector  
deleting from the heap

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

MCS 360 Lecture 26  
Introduction to Data Structures  
Jan Vershelde, 22 October 2010

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector

deleting from the heap

### 2 Binary Search Trees

sorting integer numbers

deleting from a binary search tree

### 3 Huffman Trees

encoding messages

a recursive tree creation algorithm

## the Heap

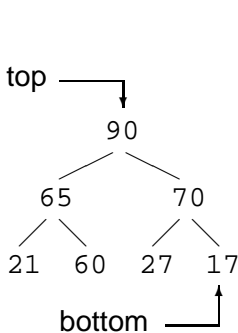
storing the heap with  
a vectordeleting from the  
heapBinary Search  
Treessorting integer  
numbersdeleting from a  
binary search tree

## Huffman Trees

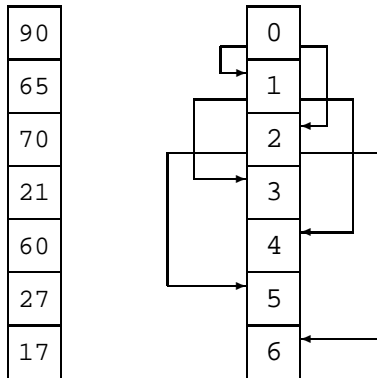
encoding messages

a recursive tree  
creation algorithm

A heap is a binary tree:



## a heap as a vector



For node at  $p$ : left child is at  $2p + 1$ , right child is at  $2p + 2$ .  
 Parent of node at  $p$  is at  $(p - 1)/2$ .

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector  
**deleting from the heap**

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

# popping from a heap

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

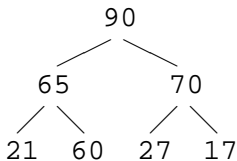
sorting integer  
numbers

deleting from a  
binary search tree

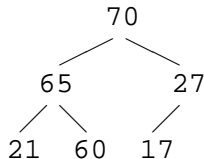
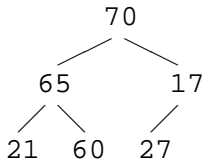
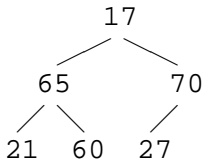
## Huffman Trees

encoding messages

a recursive tree  
creation algorithm



We replace the top first with the bottom,  
and then swap as long as parent less than largest child:



## the class Heap

## the Heap

storing the heap with  
a vector

deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

```
class Heap
{
    private:

        std::vector<int> h;

        int index_to_bottom;

        int max_child( int p );
        // returns index of largest
        // child or -1 if no child

        void swap_from_top( int p );
        // swaps the element at p with
        // its largest child if that child
        // has value larger than the parent
```

## largest child

## the Heap

storing the heap with  
a vector

deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

```
int Heap::max_child( int p )
{
    if(index_to_bottom <= p)
        return -1;
    else {
        int left = 2*p+1;
        int right = 2*p+2;
        if(left > index_to_bottom)
            return -1;
        else {
            if(right > index_to_bottom)
                return left;
            else
                return (h[left] > h[right]) ?
                    left : right;
        }
    }
}
```

## the Heap

storing the heap with  
a vector

deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

```
void Heap::pop() {
    if(index_to_bottom == 0)
        index_to_bottom--;
    else {
        h[0] = h[index_to_bottom--];
        swap_from_top(0);
    }
}

void Heap::swap_from_top( int p ) {
    if(index_to_bottom == -1) return;
    int c = max_child(p);
    if(c == -1) return;
    if(h[c] > h[p]) {
        int t = h[p];
        h[p] = h[c]; h[c] = t;
        swap_from_top(c);
    }
}
```

## the Heap

storing the heap with  
a vector

deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

```
void Heap::pop() {
    if(index_to_bottom == 0)
        index_to_bottom--;
    else {
        h[0] = h[index_to_bottom--];
        swap_from_top(0);
    }
}

void Heap::swap_from_top( int p ) {
    if(index_to_bottom == -1) return;
    int c = max_child(p);
    if(c == -1) return;
    if(h[c] > h[p]) {
        int t = h[p];
        h[p] = h[c]; h[c] = t;
        swap_from_top(c);
    }
}
```

22 Oct 2010

# sorting with a heap

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

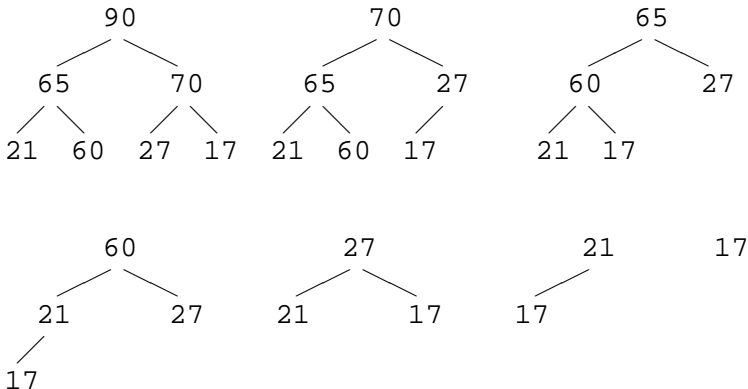
sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm



Terminology: a heap is a *complete* binary tree.

# the STL `priority_queue`

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

A heap implements a *priority queue*.

```
#include <queue>

using namespace std;

int main()
{
    priority_queue<int> q;
```

# push( ), top( ), and pop( )

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

Pushing  $n$  random numbers:

```
for(int i=0; i<n; i++)  
{  
    int r = 10+rand() % 90;  
    q.push(r);  
}
```

Sorting with top and pop:

```
vector<int> result;  
for(; q.size() > 0; q.pop())  
    result.push_back(q.top());
```

The numbers in `result` are in decreasing order.

# push( ), top( ), and pop( )

## the Heap

storing the heap with  
a vector

deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Pushing  $n$  random numbers:

```
for(int i=0; i<n; i++)  
{  
    int r = 10+rand() % 90;  
    q.push(r);  
}
```

Sorting with top and pop:

```
vector<int> result;  
for(; q.size() > 0; q.pop())  
    result.push_back(q.top());
```

The numbers in `result` are in decreasing order.

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector  
deleting from the heap

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

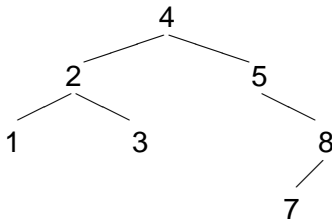
### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

# Sorting Numbers using a Tree

Consider the sequence 4, 5, 2, 3, 8, 1, 7

Insert the numbers in a tree:



Rules to insert  $x$  at node  $N$ :

- if  $N$  is empty, then put  $x$  in  $N$
- if  $x < N$ , insert  $x$  to the left of  $N$
- if  $x \geq N$ , insert  $x$  to the right of  $N$

Recursive printing: left, node, right sorts the sequence.

# finding smallest element

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

## Recursive algorithm to find smallest element?

```
Tree Tree::smallest() const
{
    if(root == NULL)
        return NULL;
    else if(root->left == NULL)
        return Tree(root);
    else
    {
        Tree L = this->get_left();
        return L.smallest();
    }
}
```

# finding smallest element

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Recursive algorithm to find smallest element?

```
Tree Tree::smallest() const
{
    if(root == NULL)
        return NULL;
    else if(root->left == NULL)
        return Tree(root);
    else
    {
        Tree L = this->get_left();
        return L.smallest();
    }
}
```

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector  
deleting from the heap

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

## deleting the root

Assume the binary search tree is not empty.  
We want to delete the root node.

- right or left child is null:



→ removing root  $r$  leads to subtree  $S$

- left or right child is null:

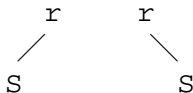


- otherwise ...

## deleting the root

Assume the binary search tree is not empty.  
We want to delete the root node.

- right or left child is null:



→ removing root  $r$  leads to subtree  $S$

- left or right child is null:

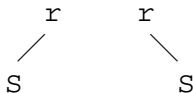


- otherwise ...

## deleting the root

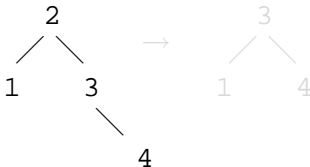
Assume the binary search tree is not empty.  
We want to delete the root node.

- right or left child is null:



→ removing root  $r$  leads to subtree  $S$

- left or right child is null:

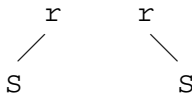


- otherwise ...

## deleting the root

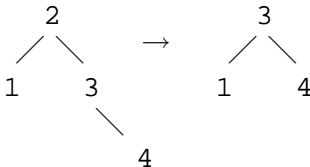
Assume the binary search tree is not empty.  
We want to delete the root node.

- right or left child is null:



→ removing root  $r$  leads to subtree  $S$

- left or right child is null:



- otherwise ...

## otherwise ... the general case

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

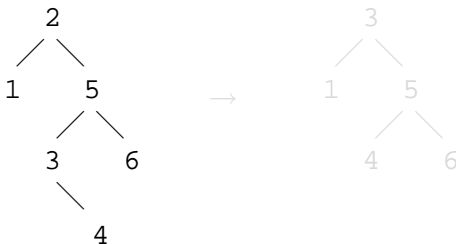
sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Removing the root:



Algorithm:

- ① Find the smallest element of the right child.
- ② Replace the root with that smallest element.
- ③ Update the parent of node of smallest element.

Observe that the smallest element has no left child.

## otherwise ... the general case

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

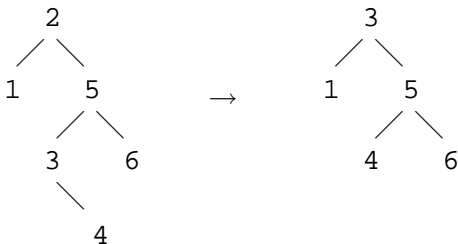
sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Removing the root:



Algorithm:

- ① Find the smallest element of the right child.
- ② Replace the root with that smallest element.
- ③ Update the parent of node of smallest element.

Observe that the smallest element has no left child.

## otherwise ... the general case

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

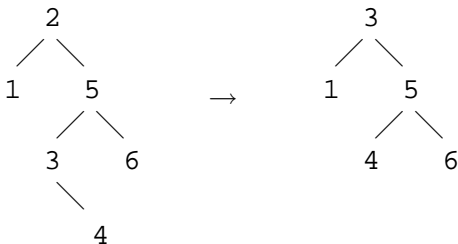
sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Removing the root:



Algorithm:

- ① Find the smallest element of the right child.
- ② Replace the root with that smallest element.
- ③ Update the parent of node of smallest element.

Observe that the smallest element has no left child.

# finding the parent node

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

We have already code to find the smallest element.  
Needed: find the parent of the smallest node.

The base cases:

```
Node* Tree::find_parent_node(int item) const
{
    if(root == NULL)
        return NULL;
    else if(root->data == item)
        return NULL;
    else
```

# finding the parent node

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

We have already code to find the smallest element.  
Needed: find the parent of the smallest node.

The base cases:

```
Node* Tree::find_parent_node(int item) const
{
    if(root == NULL)
        return NULL;
    else if(root->data == item)
        return NULL;
    else
```

## finding the parent in general

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers

deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

```
bool found = false;
Node *parent = root;
Node *child;
do
{
    if(item < parent->data)
        child = parent->left;
    else
        child = parent->right;
    if(child == NULL)
        return NULL;
    else if(child->data == item)
        found = true;
    else
        parent = child;
}
while(!found);
return parent;
```

the `remove()` method

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

```
void Tree::remove(int item)
{
    if(root == NULL)
        return;
    else if(root->data == item)
    {
        if(root->left == NULL)
            root = root->right;
        else if(root->right == NULL)
            root = root->left;
        else if(root->right->left == NULL)
        {
            root->right->left = root->left;
            root = root->right;
        }
    }
}
```

## remove ( ) continued

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

```

else
{
    Tree R = this->get_right();
    Tree L = R.get_left();
    Tree S = L.smallest();
    int md = S.get_data();

    root->data = md;
    Node *p = R.find_parent_node(md);
    Node *r = NULL;
    if(!S.is_right_null()) r = S.get_right().root;

    if(p->left == NULL)
        p->right = r;
    else if(p->left->data == md)
        p->left = r;
    else
        p->right = r;
}

```

# removing any item

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Knowing the removal of the root of a binary search tree,  
can we work with a “local root”?

Given an item that occurs in the search tree:

- 1 Find the item and its parent.
- 2 Consider the item as a “local root” node.
- 3 Update the appropriate child of the parent  
with the tree that has the item removed.

Why does this approach work?

# removing any item

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Knowing the removal of the root of a binary search tree, can we work with a “local root”?

Given an item that occurs in the search tree:

- 1 Find the item and its parent.
- 2 Consider the item as a “local root” node.
- 3 Update the appropriate child of the parent with the tree that has the item removed.

Why does this approach work?

# removing any item

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Knowing the removal of the root of a binary search tree, can we work with a “local root”?

Given an item that occurs in the search tree:

- 1 Find the item and its parent.
- 2 Consider the item as a “local root” node.
- 3 Update the appropriate child of the parent with the tree that has the item removed.

Why does this approach work?

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector  
deleting from the heap

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

## binary codes

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

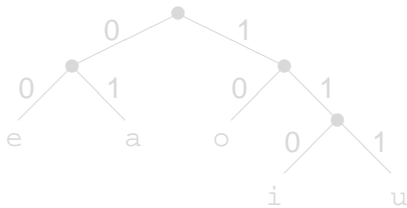
sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

A Huffman tree is a binary tree with data at leaves.  
Turn left: add 0, turn right: add 1 to code.

Vowels e, a, o are more frequent than o and u.



binary code	
e	0 0
a	0 1
o	1 0
i	1 1 0
u	1 1 1

Decode bit string by walking the tree, encode: use table.

## binary codes

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

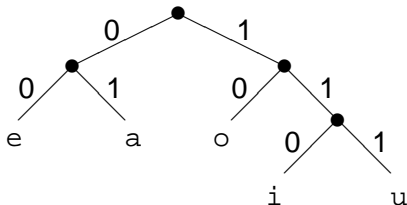
sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

A Huffman tree is a binary tree with data at leaves.  
Turn left: add 0, turn right: add 1 to code.

Vowels e, a, o are more frequent than o and u.



binary code	
e	0 0
a	0 1
o	1 0
i	1 1 0
u	1 1 1

Decode bit string by walking the tree, encode: use table.

## encoding messages

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

$A$  is the alphabet  $\{a_1, a_2, \dots, a_n\}$  of symbols,  
a message  $M$  is a sequence of elements of  $A$ .

$f_M(a_i)$  is the frequency of the symbol  $a_i$  occurring in  $M$

$$\begin{aligned} d_H(a_i) &= \text{depth of } a_i \text{ in Huffman tree} \\ &= \text{\#bits in encoding of } a_i \end{aligned}$$

\#bits of  $M$  as encoded by Huffman tree  $H$ :

$$|M|_H = \sum_{a \in A} f_M(a) \times d_H(a)$$

Goal: find optimal Huffman tree  $H$  for message  $M$ .

## encoding messages

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

$A$  is the alphabet  $\{a_1, a_2, \dots, a_n\}$  of symbols,

a message  $M$  is a sequence of elements of  $A$ .

$f_M(a_i)$  is the frequency of the symbol  $a_i$  occurring in  $M$

$$\begin{aligned} d_H(a_i) &= \text{depth of } a_i \text{ in Huffman tree} \\ &= \text{\#bits in encoding of } a_i \end{aligned}$$

#bits of  $M$  as encoded by Huffman tree  $H$ :

$$|M|_H = \sum_{a \in A} f_M(a) \times d_H(a)$$

Goal: find optimal Huffman tree  $H$  for message  $M$ .

## encoding messages

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

$A$  is the alphabet  $\{a_1, a_2, \dots, a_n\}$  of symbols,

a message  $M$  is a sequence of elements of  $A$ .

$f_M(a_i)$  is the frequency of the symbol  $a_i$  occurring in  $M$

$$\begin{aligned} d_H(a_i) &= \text{depth of } a_i \text{ in Huffman tree} \\ &= \text{\#bits in encoding of } a_i \end{aligned}$$

\#bits of  $M$  as encoded by Huffman tree  $H$ :

$$|M|_H = \sum_{a \in A} f_M(a) \times d_H(a)$$

Goal: find optimal Huffman tree  $H$  for message  $M$ .

# Priority Queues and Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

### 1 the Heap

storing the heap with a vector  
deleting from the heap

### 2 Binary Search Trees

sorting integer numbers  
deleting from a binary search tree

### 3 Huffman Trees

encoding messages  
a recursive tree creation algorithm

## a recursive idea

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

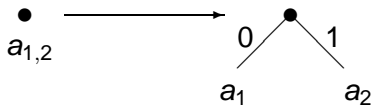
For two symbols,  $n = 2$ :  $a_1 \mapsto 0$ ,  $a_2 \mapsto 1$ .

Order symbols along their frequencies in message  $M$ :

$$f_M(a_1) \leq f_M(a_2) \leq f_M(a_3) \leq \dots \leq f_M(a_n)$$

Replace  $a_1$  and  $a_2$  by  $a_{1,2}$ ,  $f_M(a_{1,2}) = f_M(a_1) + f_M(a_2)$ ,  
then  $M_{n-1}$  is the message  $M$  over  $\{a_{1,2}, a_3, \dots, a_n\}$   
and let  $H_{n-1}$  be the optimal Huffman tree to encode  $M_{n-1}$ .

A Huffman tree  $H$  for  $M$  is then obtained via



Claim: this  $H$  obtained recursively is optimal for  $M$ .

22 Oct 2010

## running the algorithm

## the Heap

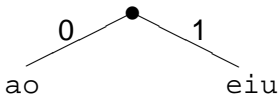
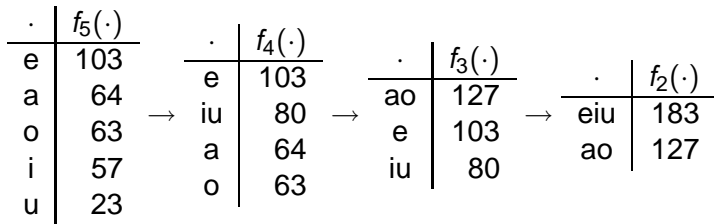
storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm



22 Oct 2010

## running the algorithm

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

Binary Search  
Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

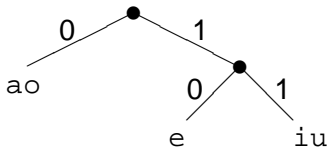
·	$f_5(\cdot)$
e	103
a	64
o	63
i	57
u	23

→

·	$f_4(\cdot)$
e	103
iu	80
a	64
o	63

→

·	$f_3(\cdot)$
ao	127
e	103
iu	80



22 Oct 2010

## running the algorithm

## the Heap

storing the heap with  
a vectordeleting from the  
heapBinary Search  
Treessorting integer  
numbersdeleting from a  
binary search tree

## Huffman Trees

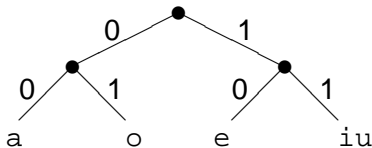
encoding messages

a recursive tree  
creation algorithm

·	$f_5(\cdot)$
e	103
a	64
o	63
i	57
u	23

→

·	$f_4(\cdot)$
e	103
iu	80
a	64
o	63



22 Oct 2010

## running the algorithm

## the Heap

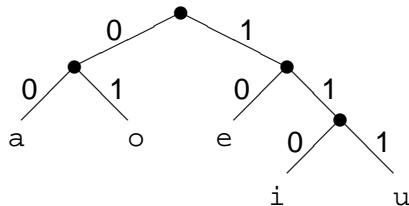
storing the heap with  
a vectordeleting from the  
heapBinary Search  
Treessorting integer  
numbersdeleting from a  
binary search tree

## Huffman Trees

encoding messages

a recursive tree  
creation algorithm

$\cdot$	$f_5(\cdot)$
e	103
a	64
o	63
i	57
u	23



# Notes on Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

A Huffman tree is a *full* binary tree:  
every node has two nonempty children.

Optimal Huffman trees are not unique.

Elements of an algorithm for Huffman Trees:

- Make frequency table of symbols in a text.  
Nodes in priority queue are of type `struct` containing  
`string` for the symbol and `int` for count.
- Recursive tree creation algorithm.  
Going forward: contract the frequency table.  
Returns from the recursive calls refines the tree.

Symbols occurring with least frequency will appear  
separately only in the last stage of the algorithm.

# Notes on Huffman Trees

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

A Huffman tree is a *full* binary tree:  
every node has two nonempty children.

Optimal Huffman trees are not unique.

Elements of an algorithm for Huffman Trees:

- Make frequency table of symbols in a text.  
Nodes in priority queue are of type `struct` containing  
`string` for the symbol and `int` for count.
- Recursive tree creation algorithm.  
Going forward: contract the frequency table.  
Returns from the recursive calls refines the tree.

Symbols occurring with least frequency will appear  
separately only in the last stage of the algorithm.

# Summary + Assignments

## the Heap

storing the heap with  
a vector  
deleting from the  
heap

## Binary Search Trees

sorting integer  
numbers  
deleting from a  
binary search tree

## Huffman Trees

encoding messages  
a recursive tree  
creation algorithm

Ended chapter 8 on trees.

## Assignments:

- 1 Write an iterative version of the `smallest()` method to return the smallest element in a binary search tree.
- 2 Give code to use a STL priority queue to create a frequency table for lower case letters in a string.
- 3 Define a binary search tree for strings and use it to store words that appear in a text on file. A word is separated by one or more spaces.
- 4 Define the algorithm to decode a message (given as bit string) using a Huffman tree.