

heaps and Huffman codes

priority queues: motivation

dynamically changing list of events with dates

want to find next event quickly

list of running programs, some more important (e.g. what user will notice being slow)

choose most important to run first

want to find most important quickly

list of connections, some interactive (video call), some not (download)

want quick way to choose which one to service

data structure: priority queue

priority queue ADT

`insert(priority, item)`

`findMin()` — return item with lowest (first) priority

`deleteMin()` — remove item with lowest (first) priority

priority queue implementations

structure	insert	findMin	deleteMin
unsorted vector	$\Theta(1)$ (amortized)	$\Theta(n)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	$\Theta(1)$	amortized $\Theta(\log n)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

priority queue implementations

structure	insert	findMin	deleteMin
unsorted vector	$\Theta(1)$ (amortized)	$\Theta(n)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	$\Theta(1)$	amortized $\Theta(\log n)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

priority queue implementations

structure	insert	findMin	deleteMin
unsorted vector	$\Theta(1)$ (amortized)	$\Theta(n)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	$\Theta(1)$	amortized $\Theta(\log n)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$

additional, optional operations

not necessary to have a priority queue, but useful...

decreaseKey — change value of key given index/pointer

remove — remove value with given index/pointer

structure	decreaseKey	remove
unsorted vector	$\Theta(1)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(n)$
sorted linked list	$\Theta(n)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	amortized $\Theta(1)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$

additional, optional operations

not necessary to have a priority queue, but useful...

decreaseKey — change value of key given index/pointer

remove — remove value with given index/pointer

structure	decreaseKey	remove
unsorted vector	$\Theta(1)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(n)$
sorted linked list	$\Theta(n)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	amortized $\Theta(1)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$

additional, optional operations

not necessary to have a priority queue, but useful...

decreaseKey — change value of key given index/pointer

remove — remove value with given index/pointer

structure	decreaseKey	remove
unsorted vector	$\Theta(1)$	$\Theta(n)$
unsorted linked list	$\Theta(1)$	$\Theta(n)$
sorted vector	$\Theta(n)$	$\Theta(n)$
sorted linked list	$\Theta(n)$	$\Theta(1)$
balanced tree	$\Theta(\log n)$	$\Theta(\log n)$
binary heap	$\Theta(\log n)$	$\Theta(\log n)$
Fibannoci heap	amortized $\Theta(1)$	amortized $\Theta(1)$
strict Fibannoci heap	$\Theta(1)$	$\Theta(1)$

aside: min v max

can also have ADT with findMax/etc. *instead of* findMin/etc.

same complexities, etc. (use different comparisons)

terms for heaps: “min-heap” (findMin version) or “max-heap”
(findMax version)

binary heaps

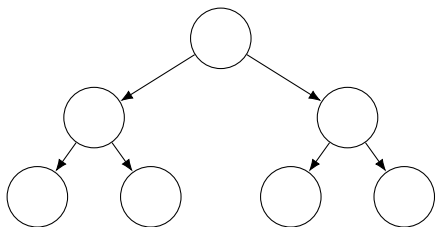
binary heap is a binary tree

binary tree is **not a binary search tree**

structure: almost a perfect tree

ordering: parent $<$ child (everywhere in tree)

perfect binary trees



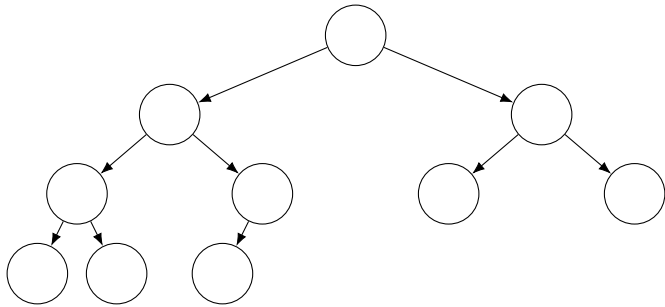
a binary tree is **perfect** or **complete** if

all leaves have same depth

all nodes have zero children (leaf) or two children

exactly the trees that achieve $2^h - 1$ nodes

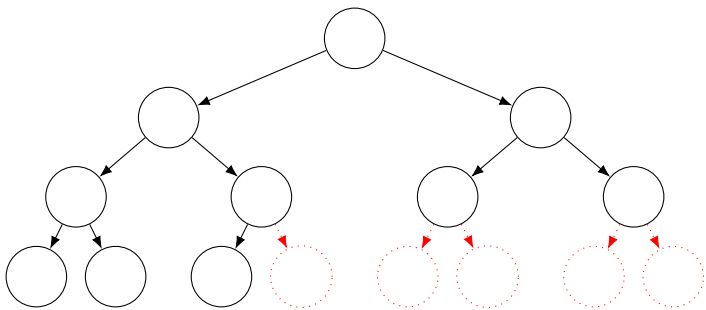
almost perfect/complete binary trees



heaps are **almost complete** trees

only missing bottom-rightmost slots

almost perfect/complete binary trees



heaps are almost complete trees

only missing bottom-rightmost slots

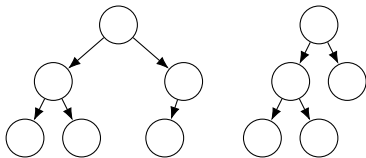
almost complete formally

single node tree is almost complete

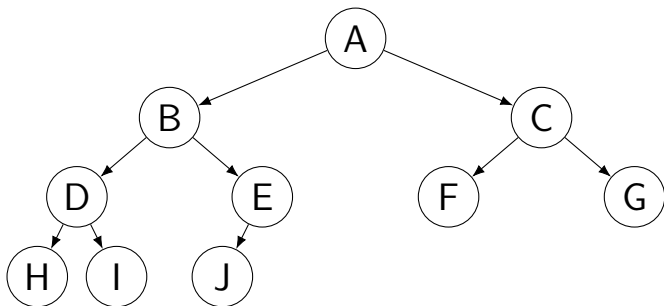
otherwise: almost complete if either

left child is complete with height h and right child almost complete with height h ; *OR*

left child is almost complete with height h and right child is complete with height $h - 1$



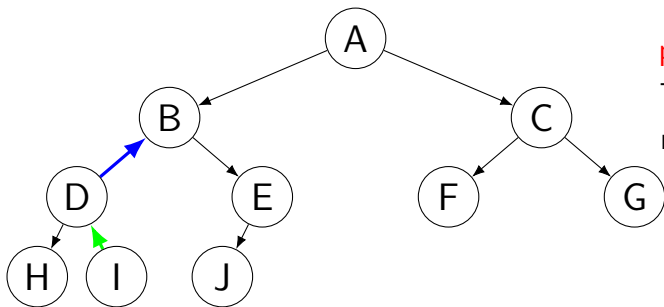
trees as arrays



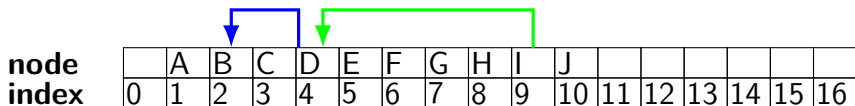
node	A	B	C	D	E	F	G	H	I	J						
index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

string theTree[17] = {"", "A", "B", ...}

trees as arrays

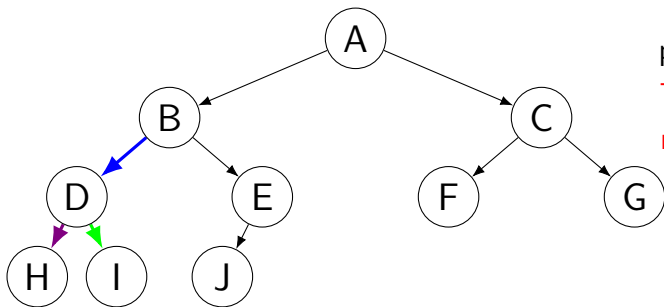


$\text{parentIndex} = \text{index} / 2$
 $\text{leftChild} = \text{index} * 2$
 $\text{rightChild} = \text{index} * 2 + 1$

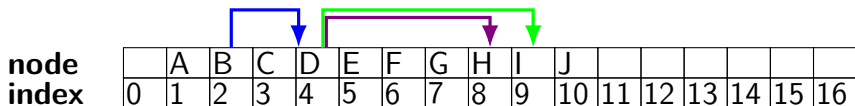


string theTree[17] = {"", "A", "B", ...}

trees as arrays



parentIndex = index / 2
leftChild = index * 2
rightChild = index * 2 + 1



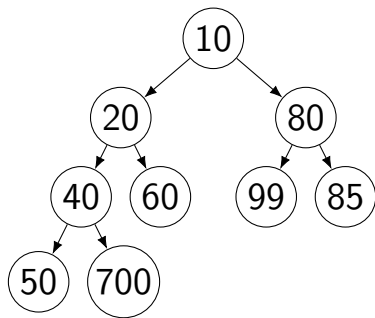
string theTree[17] = {"", "A", "B", ...}

why arrays

single array — less storage/memory allocation

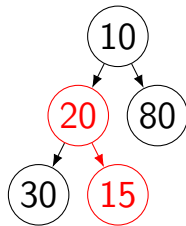
represent tree as single vector

the heap property



heap property: parent \leq any of its children

a non-heap



heap property: parent \leq any of its children

heap code

linked off slides page of repo

```
class binary_heap {  
    ...  
private:  
    // heap[1] is root  
    // leftChildIndex = index * 2  
    // rightChildIndex = index * 2 + 1  
    // parentIndex = index / 2  
    vector<int> heap;  
    int heap_size;  
}
```

heap insert

add new node as leaf node

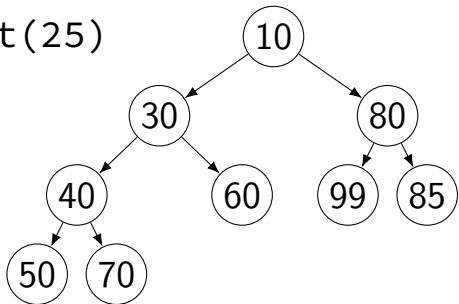
while new node $<$ parent node: swap with parent

heap insert

add new node as leaf node

while new node < parent node: swap with parent

insert(25)

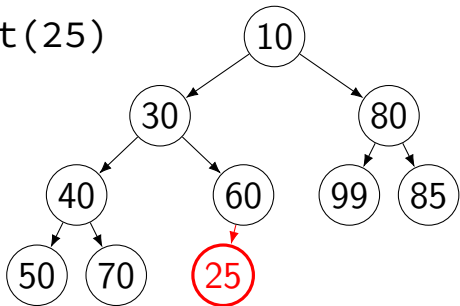


heap insert

add new node as leaf node

while new node < parent node: swap with parent

insert(25)

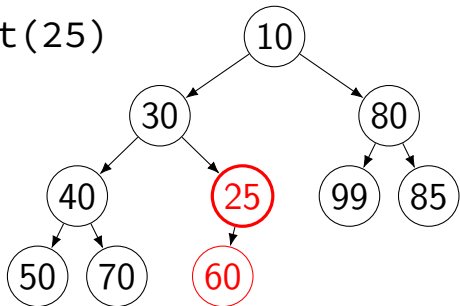


heap insert

add new node as leaf node

while new node < parent node: swap with parent

insert(25)

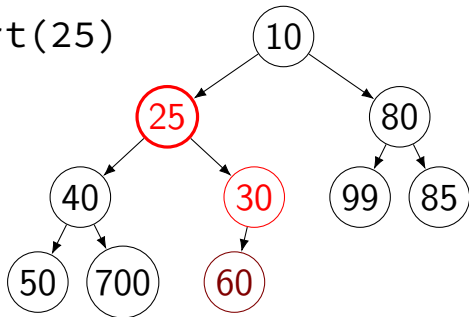


heap insert

add new node as leaf node

while new node < parent node: **swap with parent**

insert(25)



insert(int)

```
void binary_heap::insert(int x) {  
    ++heap_size;  
    heap.push_back(x);  
    percolateUp(x);  
}
```

percolateUp(int)

```
void binary_heap::percolateUp(int index) {
    int newValue = heap[index];
    // while not at root and
    //     less than parent...
    while (index > 1 && newValue < heap[index / 2]) {
        // move parent down
        heap[index] = heap[index / 2];
        // advance up the tree
        index /= 2;
    }
    heap[index] = newValue;
}
```

insert runtime

worst case: $\log_2 N$ nodes changed

insert average case?

average case is better assuming random keys:

intuition: leafs have bottom half of values (on average)

...so usually don't need to move up

...and if we do, parents of leafs have 25th to 50th percentile of values

...so need to move up two steps even less

about 2 steps moved up on average

heap deleteMin

replace root with last leaf node

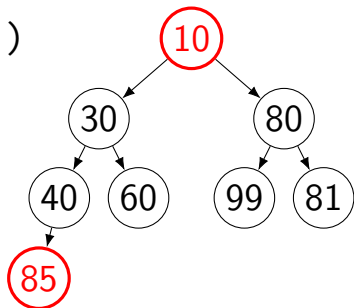
while node greater than children: swap with *smallest child*

heap deleteMin

replace root with last leaf node

while node greater than children: swap with *smallest child*

deleteMin()

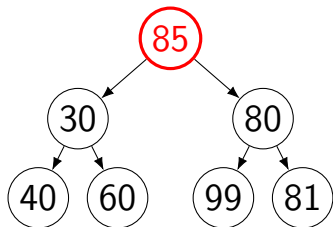


heap deleteMin

replace root with last leaf node

while node greater than children: swap with *smallest child*

deleteMin()



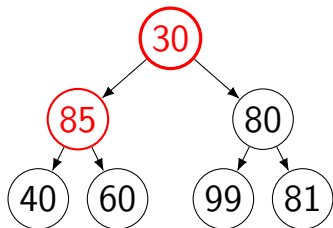
X

heap deleteMin

replace root with last leaf node

while node greater than children: *swap with smallest child*

deleteMin()

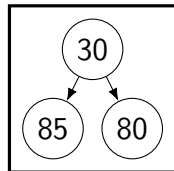
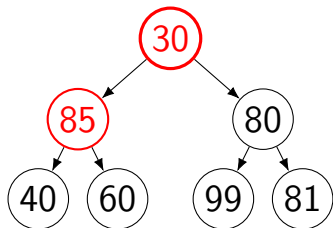


heap deleteMin

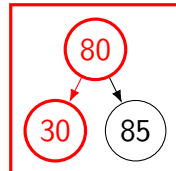
replace root with last leaf node

while node greater than children: swap with *smallest child*

deleteMin()



is a heap



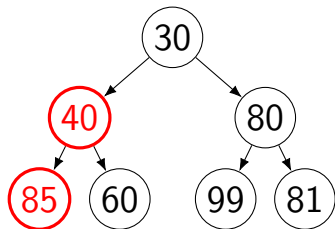
not a heap

heap deleteMin

replace root with last leaf node

while node greater than children: *swap with smallest child*

deleteMin()



deleteMin code

```
int binary_heap::deleteMin() {  
    if (heap_size == 0)  
        throw ...;  
    int result = heap[1];  
    heap[1] = heap[heap_size--];  
    heap.pop_back();  
    percolateDown(1);  
    return result;  
}
```

precolateDown code

```
int binary_heap::percolateDown(int index) {
    int value = heap[index];
    // while left child exists
    while (index * 2 <= heap_size) {
        int left = index * 2, right = index * 2 + 1;

        // set child to smallest child that exists
        int child = left;
        if (right <= heap_size && heap[right] < heap[left])
            child = right;

        // if less than smallest, done
        if (value < heap[child]) break;
        // otherwise:
        heap[index] = heap[child]; // move child up
        index = child;             // and traverse down
    }
    heap[index] = value;
}
```

}

deleteMin runtime

worst case $\Theta(\log N)$ — move nodes from root to leaf

other heap operations?

decreaseKey/increaseKey

change value, then percolateUp/Down

slow ($\Theta(N)$) if you have to find the value

fast ($\Theta(\log N)$) if you already know where value is

(one method: keep track of its index)

faster (amortized $\Theta(1)/\Theta(1)$) in Fibonacci/strict Fibonacci heaps

remove

decreaseKey, then deleteMin

core heap operations

insert — $\Theta(\log N)$ worst case, better on “average”

deleteMin — $\Theta(\log N)$

findMin — $\Theta(1)$

heap sort

```
void heapSort(vector<T>& values) {  
    binary_heap<T> heap;  
    for (T x : values)  
        heap.insert(x);  
    values.clear();  
    while (!heap.empty()) {  
        values.push_back(heap.deleteMin());  
    }  
}
```

$\Theta(N \log N)$ sort

can be done in place with more careful implementation
(use `values` as the max-heap's array,
place sorted elements starting at end)

mostly not as fast in practice as comparable *unstable* sorts

compression

compression

50KB webpage as 5KB download (a lot faster!)

100MB of machine code as 50MB download?

movie of 24 1MB pictures/second into 10MB/minute file?

...

lossy compression

for audio, pictures, video, *lossy compression* is common

intuition: you won't notice if we make the pixel 0.25% darker
...and it had "noise" from camera sensor, etc. anyways

idea: model human perception

write down **most important parts** of audio/image/etc.
important = noticed by humans

lossless compression

lossless compression — reproduce original file

rely on patterns

example: text file has many more 'e's than '!'

...so choose shorter encoding for 'e' than '!'

example: computer-drawn images have lots of white space

...so have a way to represent “a big white rectangle” (instead of specifying each pixel)

typical compression results

ratio = original size:final size

note: usually a compression ratio/speed tradeoff (not shown)

lossless:

- for English text or source code: about 4:1

- for CD-quality audio: about 2:1

- for photographs: about 2:1

- for computer-drawn diagrams: about 5:1 to 20:1

lossy: (making a guess at what is “close enough” in quality)

- for CD-quality audio: about 4:1

- for standard definition TV video+audio: about 1:40

a prefix code

letter	code
a	0
b	100
c	101
d	11

a prefix code

letter	code
a	0
b	100
c	101
d	11

prefix code

no code is prefix of another (no ambiguity)

shorter codes for more frequent values (hopefully)

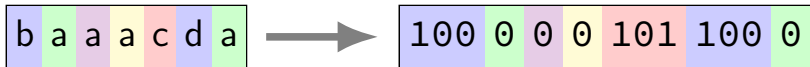
a prefix code

letter	code
a	0
b	100
c	101
d	11

prefix code

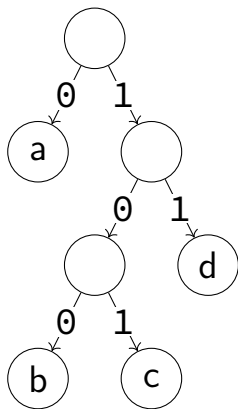
no code is prefix of another (no ambiguity)

shorter codes for more frequent values (hopefully)



prefix codes as trees

letter	code
a	0
b	100
c	101
d	11



prefix code cost

letter code frequency

a	0	5/12
b	100	1/6
c	101	1/12
d	11	1/3

$$\text{cost} = \sum_i p_i r_i = \frac{5}{12} \cdot 1 + \frac{1}{6} \cdot 3 + \frac{1}{12} \cdot 3 + \frac{1}{3} \cdot 2 = \frac{11}{6} \text{ (bits per symbol)}$$

p_i : probability symbol i occurs

r_i : length of code for i

prefix code cost

letter code frequency

a	00	5/12
b	100	1/6
c	101	1/12
d	11	1/3

$$\text{cost} = \sum_i p_i r_i = \frac{5}{12} \cdot 1 + \frac{1}{6} \cdot 3 + \frac{1}{12} \cdot 3 + \frac{1}{3} \cdot 2 = \frac{11}{6} \text{ (bits per symbol)}$$

p_i : probability symbol i occurs

r_i : length of code for i

versus a=00,b=01,c=10,d=11: cost = 2 (bits per symbol)

how to find **minimum cost prefix code** (given frequencies)?

high-level compression steps

read file, find symbol frequencies

choose best prefix code (called *Huffman code*) based on frequencies

best = assuming each code maps to one symbol

write prefix code to output

read file, convert to prefix code, write to output



high-level compression steps

read file, find symbol frequencies

choose best prefix code (called *Huffman code*) based on frequencies

best = assuming each code maps to one symbol

write prefix code to output

read file, convert to prefix code, write to output



finding the best prefix code

build prefix code tree from bottom up

intuition 1: least frequent thing at bottom \rightarrow use it first
use case for a priority queue

intuition 2: combine less frequent symbols into more frequent group
work with **partial prefix trees**

running example and frequencies

if it is to be, it is up to me

symbol	frequency	symbol	frequency
b	1	p	1
e	2	s	2
f	1	t	4
i	5	u	1
m	1	, (comma)	1
o	2	␣ (space)	9

building the Huffman tree (1)

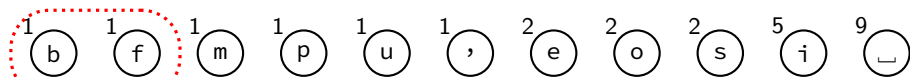


list of **partial prefix trees**

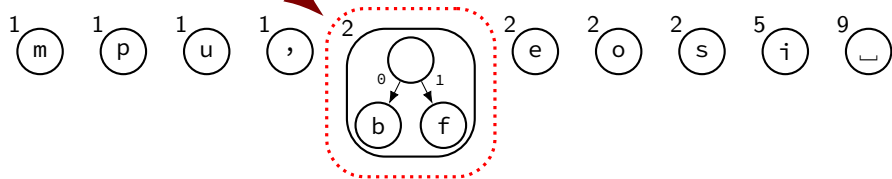
labelled with **total frequency of contained symbols**

goal: combine these into one prefix tree

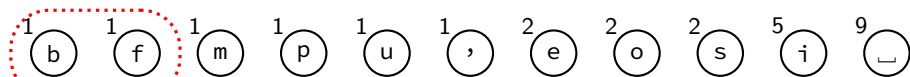
building the Huffman tree (1)



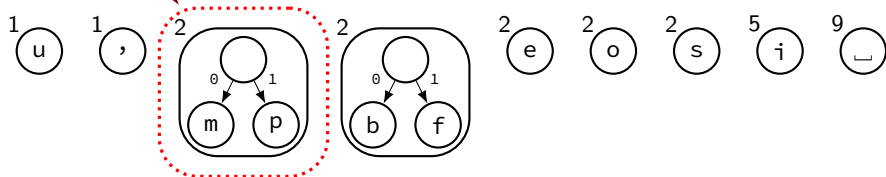
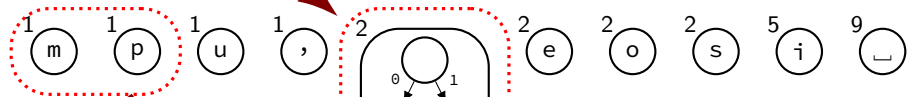
combine **two least frequent** into partial prefix tree
new frequency = sum of old frequencies



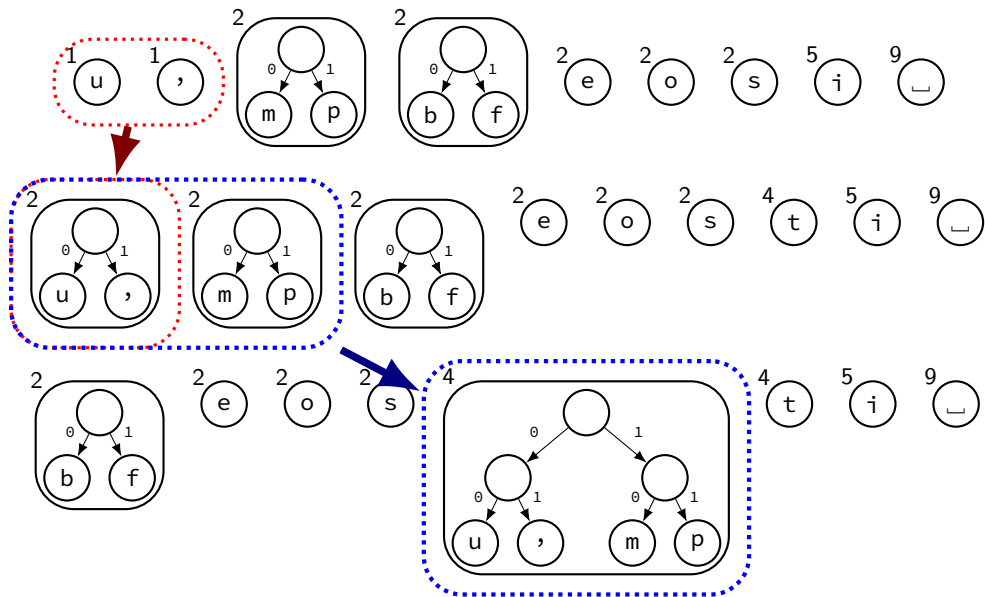
building the Huffman tree (1)



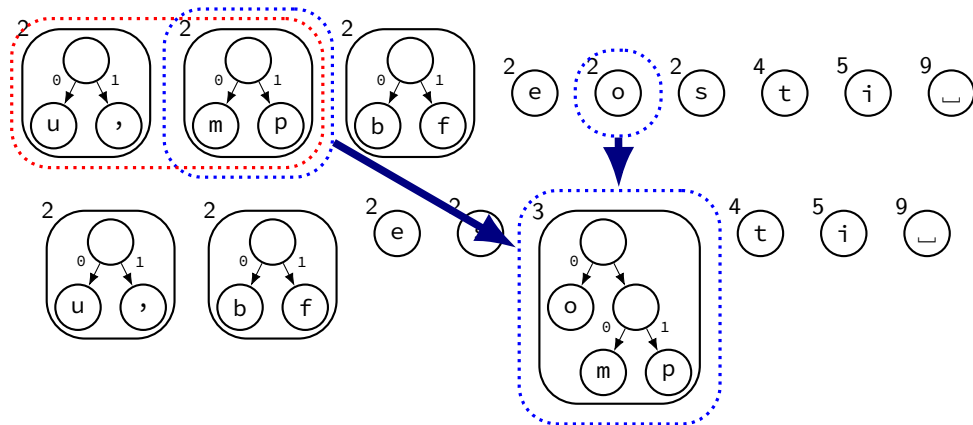
combine **two least frequent** into partial prefix tree
new frequency = sum of old frequencies



building the Huffman tree (2)

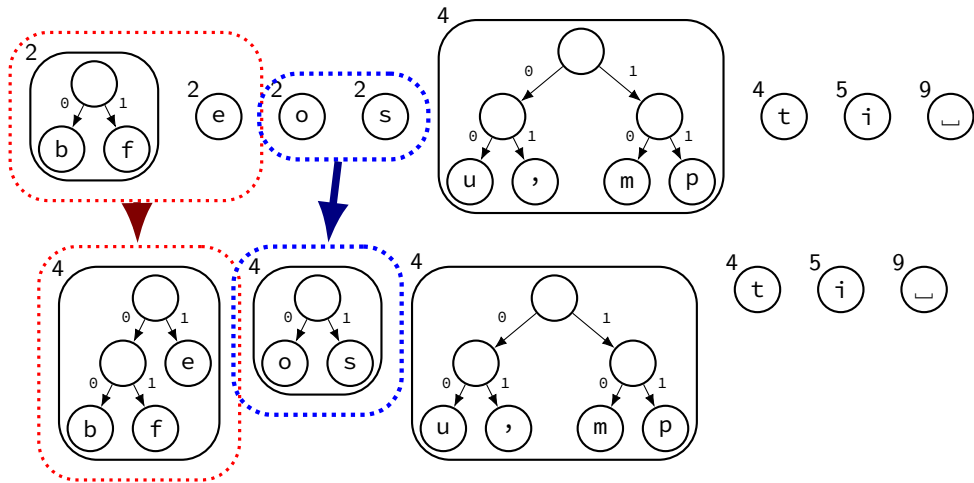


building the Huffman tree: alternatives

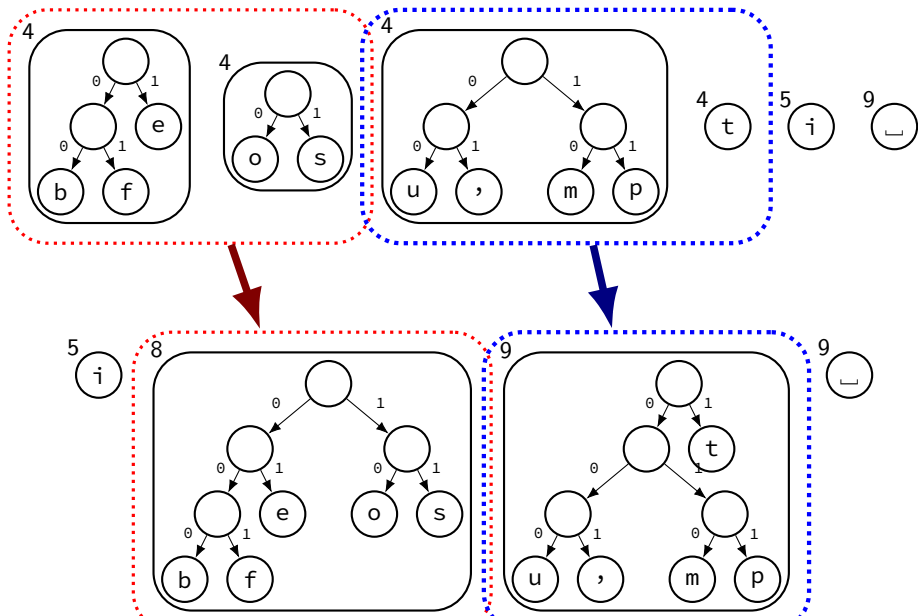


multiple choices of what to combine
proof not shown: produce same quality prefix tree

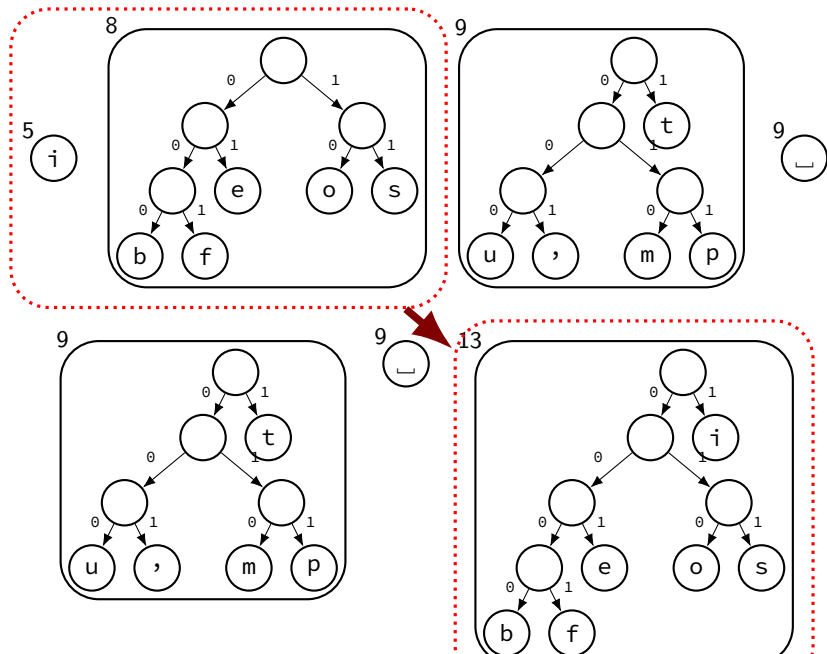
building the Huffman tree (3)



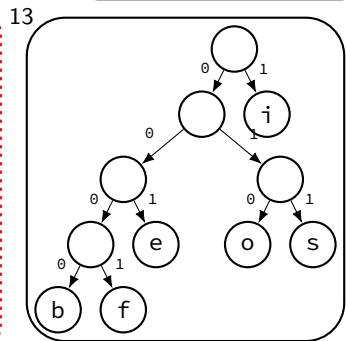
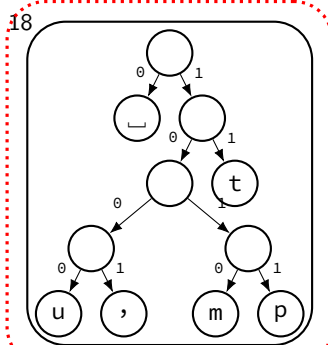
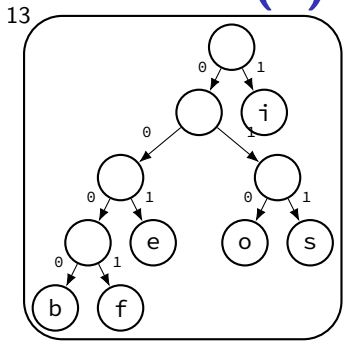
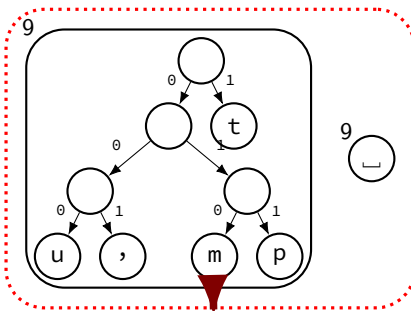
building the Huffman tree (4)



building the Huffman tree (5)

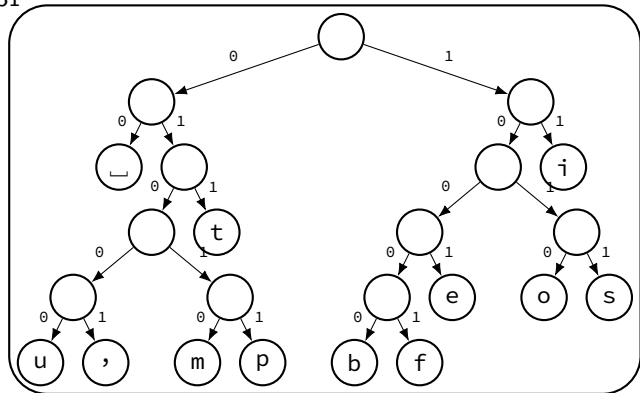


building the Huffman tree (6)

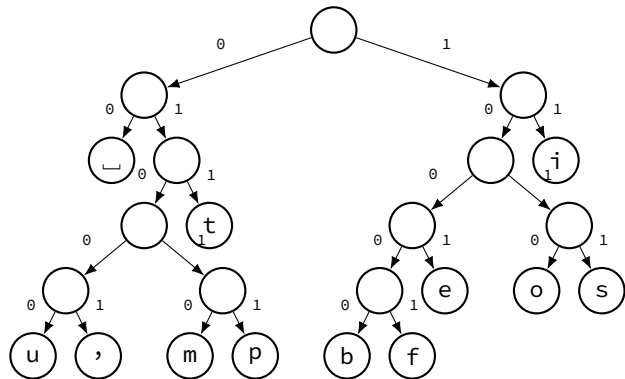


building the Huffman tree (7)

31



the final Huffman tree



letter code

l	00
u	01000
,	01001
m	01010
p	01011
t	011
b	10000
f	10001
e	1001
o	1010
s	1011
i	11

tree-building pseudocode

```
class PrefixTree {
    ...
    PrefixTree(char c, int frequency);
    PrefixTree(PrefixTree rightSide, PrefixTree leftSide);
    PrefixTree(const PrefixTree &other);
    ...
};
...
PriorityQueue<PrefixTree> queue;
for (char c, frequency f in inputFile) {
    queue.insert(PrefixTree(c, f));
}
while (queue.size() > 1) {
    PrefixTree first = queue.deleteMin();
    PrefixTree second = queue.deleteMin();
    queue.insert(PrefixTree(first, second));
}
return queue.deleteMin();
...
```

storing the prefix code

file format for the lab:

```
space 00
u 01000
, 01001
m 01010
p 01011
t 011
b 10000
f 10001
e 1001
o 1010
s 1011
i 11
```

real format?

does this save space?

probably if input file is big enough...

but real compression formats use a more compact encoding
not having you do in lab to ease debugging/etc.

what about the data?

in lab: the text 01111110011110...

obviously wastes a lot of space...

real compression: sequence of bytes, 8 bits per

extra work to extract bit-by-bit, match with prefix code

last time

an application for trees and heaps: Huffman coding

goal: lossless compression

divide document into symbols (e.g. characters)

choose *variable length encodings* for symbols

prefix code — no code prefix of another
represented by a tree

Huffman coding — produce optimal cost prefix codes:

priority queue (ordered by frequency) of partial prefix code trees
build from bottom up, least frequent first

decoding

load the code **into a prefix code tree**

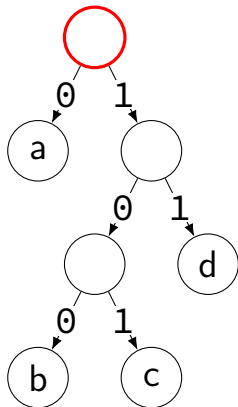
then, read bits, traversing tree until leaf

psuedocode:

```
while (there are more bits) {
    PrefixTreeNode *current = root;
    while (current is not a leaf) {
        if (next bit is 0)
            current = current->left;
        else
            current = current->right;
    }
    output(current->symbol);
}
```

example

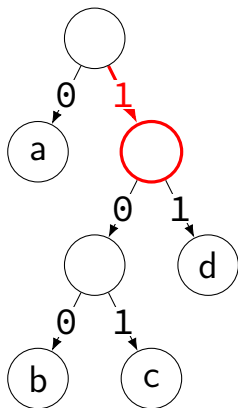
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

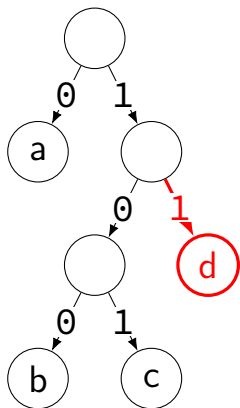
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

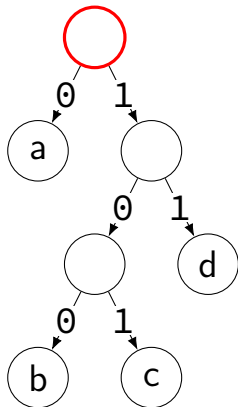
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

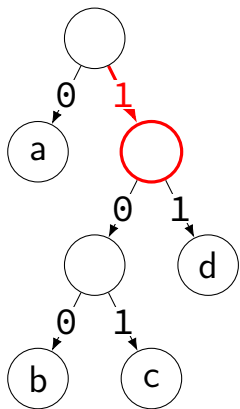
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

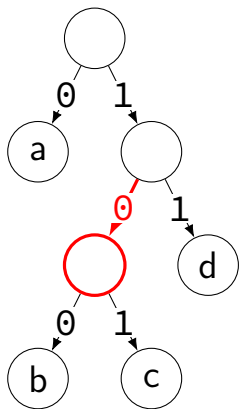
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

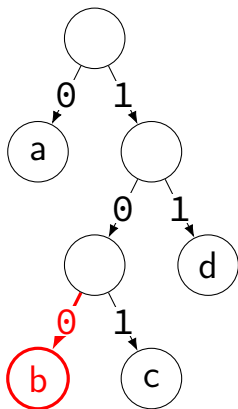
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

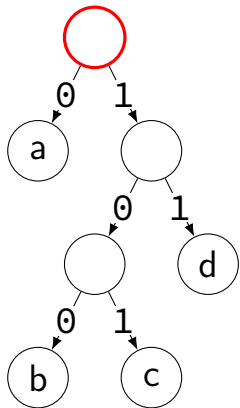
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

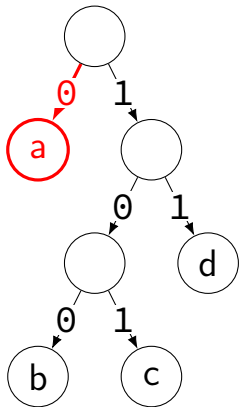
letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

example

letter	code
a	0
b	100
c	101
d	11



11 100 0 101 0 0 11 = dba caad

lab preview

pre-lab: compression

in-lab: decompression

post-lab report

pre-lab

write a program to...

calculate letter frequencies of input

use binary heap to build huffman tree

output encoding mapping (format specified in lab)

output encoded message

pre-lab tools

heap code supplied in slides

file I/O code provided (`fileio.cpp`)

or see `getWordInTable.cpp` from lab 6

or see <http://www.cplusplus.com/doc/tutorial/files/>

or see ifstream documentation

a note on ASCII

the American standard character codes

- 7-bit characters (extra bit left over in bytes)

- ASCII or superset used to represent English text

128 characters (95 printable, 33 non-printable)

Wikipedia article as [table/details](#)

ASCII codes

for lab: only worry about “printable” ASCII characters
byte values 0x20 to 0x7e

special case: 0x20 = ‘space’

no other whitespace characters used
(output character in table as itself...)

heap example

linked off slides page as

`binary_heap.h`

`binary_heap.cpp`

you may use for lab

heap declaration: public

```
class binary_heap {  
public:  
    binary_heap();  
    binary_heap(vector<int> vec);  
    ~binary_heap();  
  
    void insert(int x);  
    int findMin();  
    int deleteMin();  
    unsigned int size();  
    void makeEmpty();  
    bool isEmpty();  
    void print();  
    ...  
};
```

heap declaration: private

```
class binary_heap {  
    ...  
private:  
    vector<int> heap;  
    unsigned int heap_size;  
    void percolateUp(int hole);  
    void percolateDown(int hole);  
};
```

vector heap

`vector<int> heap` — vector representing binary tree, using rules shown before

`heap[0]` is unused

`heap[1]` is root

`heap[i * 2]` is left child of node i

`heap[i * 2 + 1]` is right child of node i

`int heap_size` is its size

(even though `heap.size() - 1` could have been used instead...)

binary_heap::binary_heap(vec)

constructor to initialize from *unsorted* vector

equivalent to repeated insertion...

```
binary_heap::binary_heap(vector<int> vec) :  
    heap_size(vec.size()) {  
    heap = vec;  
    heap.push_back(heap[0]);  
    heap[0] = 0;  
    for ( int i = heap_size/2; i > 0; i-- )  
        percolateDown(i);  
}
```

binary_heap::binary_heap(vec)

constructor to initialize from *unsorted* vector

equivalent to repeated insertion...

recall: in-place heap sort — similar to what's happening here...

```
binary_heap::binary_heap(vector<int> vec) :  
    heap_size(vec.size()) {  
    heap = vec;  
    heap.push_back(heap[0]);  
    heap[0] = 0;  
    for ( int i = heap_size/2; i > 0; i— )  
        percolateDown(i);  
}
```

findMin/size/etc.

```
int binary_heap::findMin() {
    if ( heap_size == 0 )
        throw "findMin()_called_on_empty_heap";
    return heap[1];
}
```

```
unsigned int binary_heap::size() {
    return heap_size;
}
```

```
bool binary_heap::isEmpty() {
    return heap_size == 0;
}
```

```
void binary_heap::makeEmpty() {
    heap_size = 0;
    heap[1] = (1);
}
```

print

```
void binary_heap::print() {
    cout << "(" << heap[0] << ")_";
    for ( int i = 1; i <= heap_size; i++ ) {
        cout << heap[i] << "_";
        // next line from from http://tinyurl.com/mf9tbgm
        bool isPow2 = (((i+1) & ~(i))==(i+1))? i+1 : 0;
        if ( isPow2 )
            cout << endl << "\t";
    }
    cout << endl;
}
```