IS 2610: Data Structures

Priority Queue, Heapsort, Searching

March 15, 2004

Priority Queues

- Applications require that we process records with keys in order
  - Collect records
  - Process one with largest key
  - Maybe collect more records
- Applications
  - Simulation systems (event times)
  - Scheduling (priorities)
- Priority queue: A data structure of items with keys that support two basic operations: Insert a new item; Delete the item with the largest key

Priority Queue

- Build and maintain the following operations
  - Construct the queue
  - Insert a new item
  - Delete the maximum item
  - Change the priority of an arbitrary specified item
  - Delete an arbitrary specified item
  - Join two priority queues into one large one

Priority Queue: Elementary operations

```
void PQinit(int);
int PQempty();
void PQinsert(Item);
Item PQdelmax();
```

```
#include <stdlib.h>
#include "Item.h"
static Item *pq;
static int N;
void PQinit(int maxN)
  { pq = malloc(maxN*sizeof(Item)); N = 0; }
 int PQempty()
  { return N == 0; }
void PQinsert(Item v)
  { pq[N++] = v; }
Item PQdelmax()
  { int j, max = 0;
    for (j = 1; j < N; j++)
      if (less(pq[max], pq[j])) max = j;
    exch(pq[max], pq[N-1]);
    return pq[--N];
  }
```

Heap Data Structure

- Def 9.2
  - A tree is heap-ordered if the key in each node is larger than or equal to the keys in all of that node's children (if any). Equivalently, the key in each node of a heap-ordered tree is smaller than or equal to the key in that node's parent (if any)
- Property 9.1
  - No node in a heap-ordered tree has a key larger than the key at the root.
- Heap can efficiently support the basic priority-queue operations

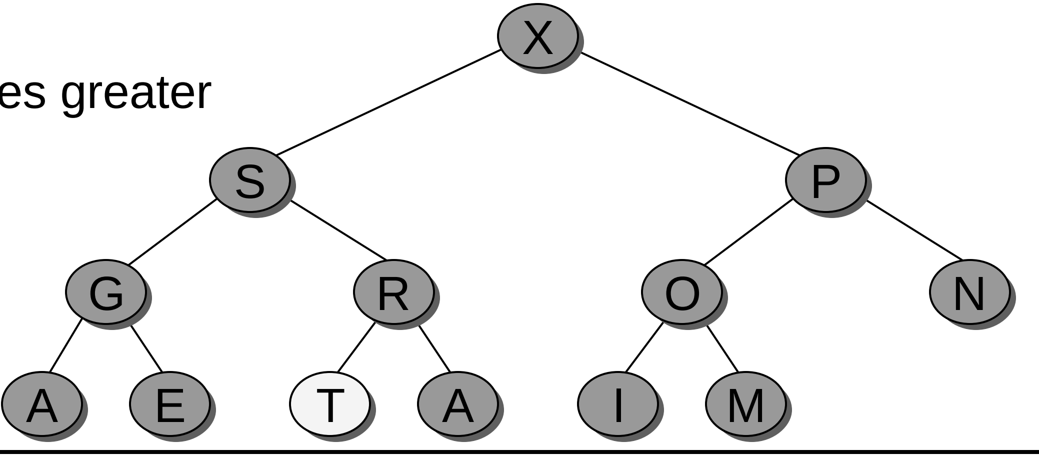
Heap Data Structure

- Def 9.2
  - A heap is a set of nodes with keys arranged in a complete heap-ordered binary tree, [represented as an array].
- A complete tree allows using a compact array representation
  - The parent of node i is in position $\lfloor i/2 \rfloor$
  - The two children of the node i are in positions $2i$ and $2i + 1$.
  - Disadvantage of using arrays?

Algorithms on Heaps

- Heapifying
  - Modify heap to violate the heap condition
    - Add new element
    - Change the priority
  - Restructure heap to restore the heap condition

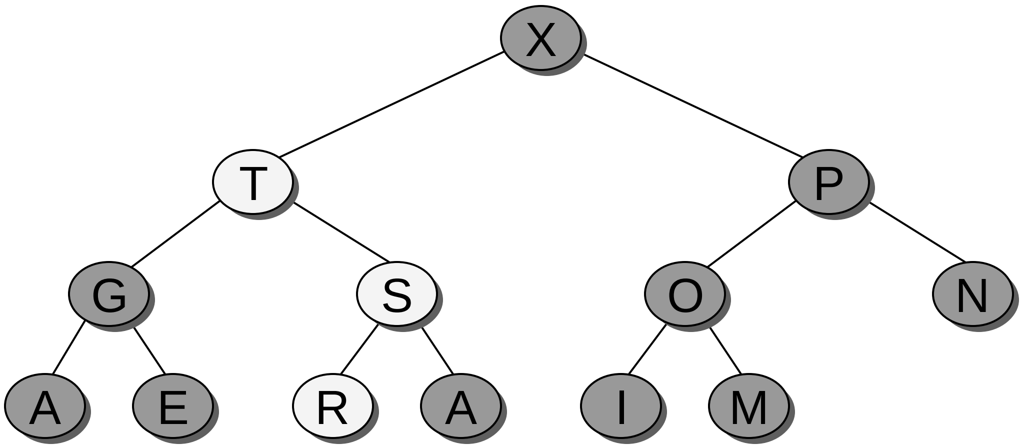
Priority of child becomes greater



Bottom-up heapify

- First exchange T and R
- Then exchange T and S

```
fixUp(Item a[], int k)
  {
    while (k > 1 && less(a[k/2], a[k]))
      { exch(a[k], a[k/2]); k = k/2; }
  }
```

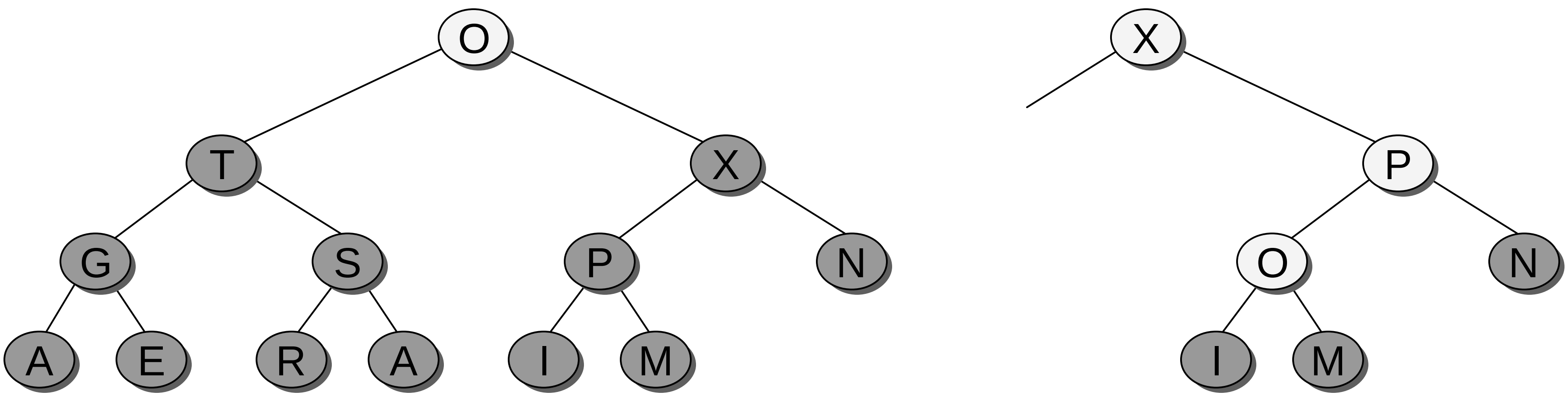


Top-down heapify

- Exchange with the larger child

```
fixDown(Item a[], int k, int N)
  { int j;
    while (2*k <= N)
      { j = 2*k;
        if (j < N && less(a[j], a[j+1])) j++;
        if (!less(a[k], a[j])) break;
        exch(a[k], a[j]); k = j;
      }
  }
```

Priority of parent becomes smaller



Heap-based priority Queue

- Property 9.2
  - Insert requires no more than $lg\ n$
    - one comparison at each level
  - Delete maximum requires no more than $2\ lg\ n$
    - two comparisons at each level

```
#include <stdlib.h>
#include "Item.h"
static Item *pq;
static int N;
void PQinit(int maxN)
  { pq = malloc((maxN+1)*sizeof(Item)); N = 0; }
 int PQempty()
  { return N == 0; }
void PQinsert(Item v)
  { pq[++N] = v; fixUp(pq, N); }
Item PQdelmax()
  {
    exch(pq[1], pq[N]);
    fixDown(pq, 1, N-1);
    return pq[N--];
  }
```

Sorting with a priority Queue

- Use PQinsert to put all the elements on the priority queue
- Use PQdelmax to remove them in decreasing order

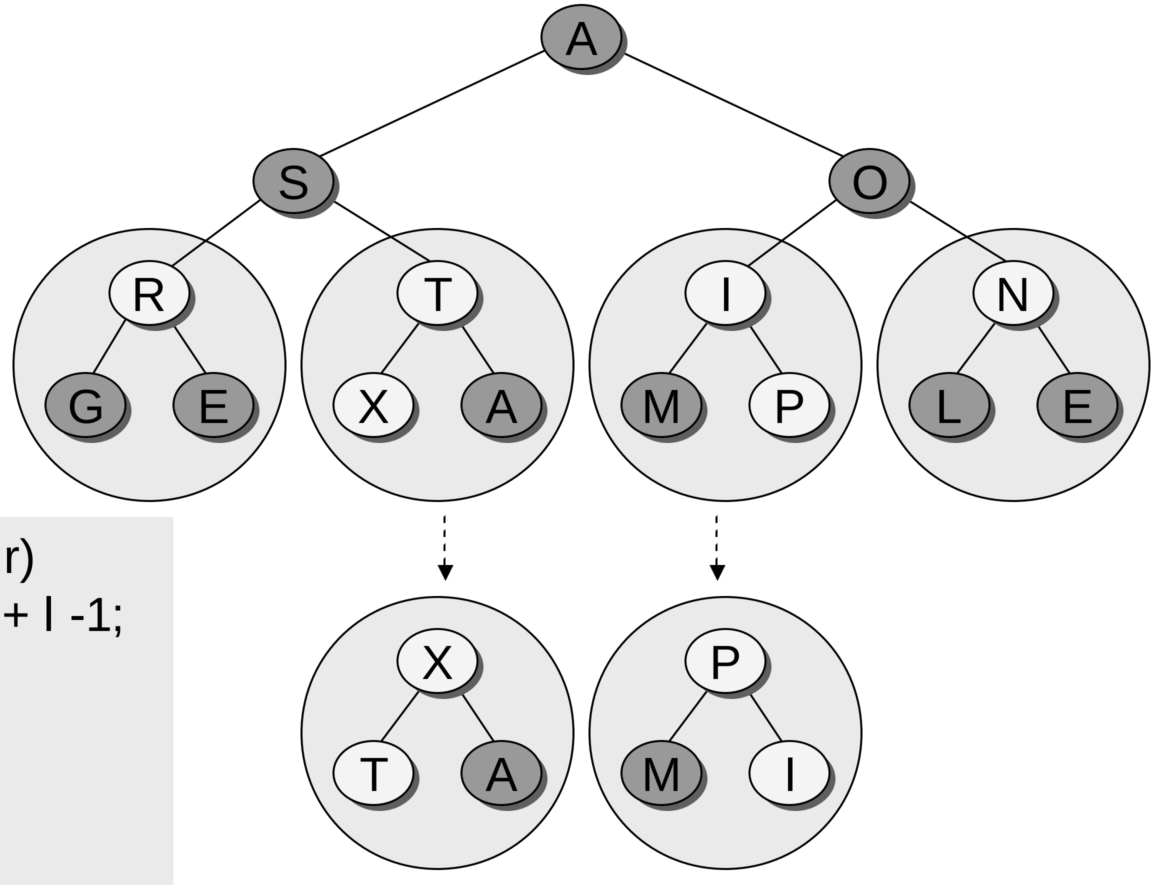
Heap construction takes

< $n \lg n$

```
void PQsort(Item a[], int l, int r)
  { int k;
    PQinit();
    for (k = l; k <= r; k++) PQinsert(a[k]);
    for (k = r; k >= l; k--) a[k] = PQdelmax();
  }
```

Bottom-up Heap

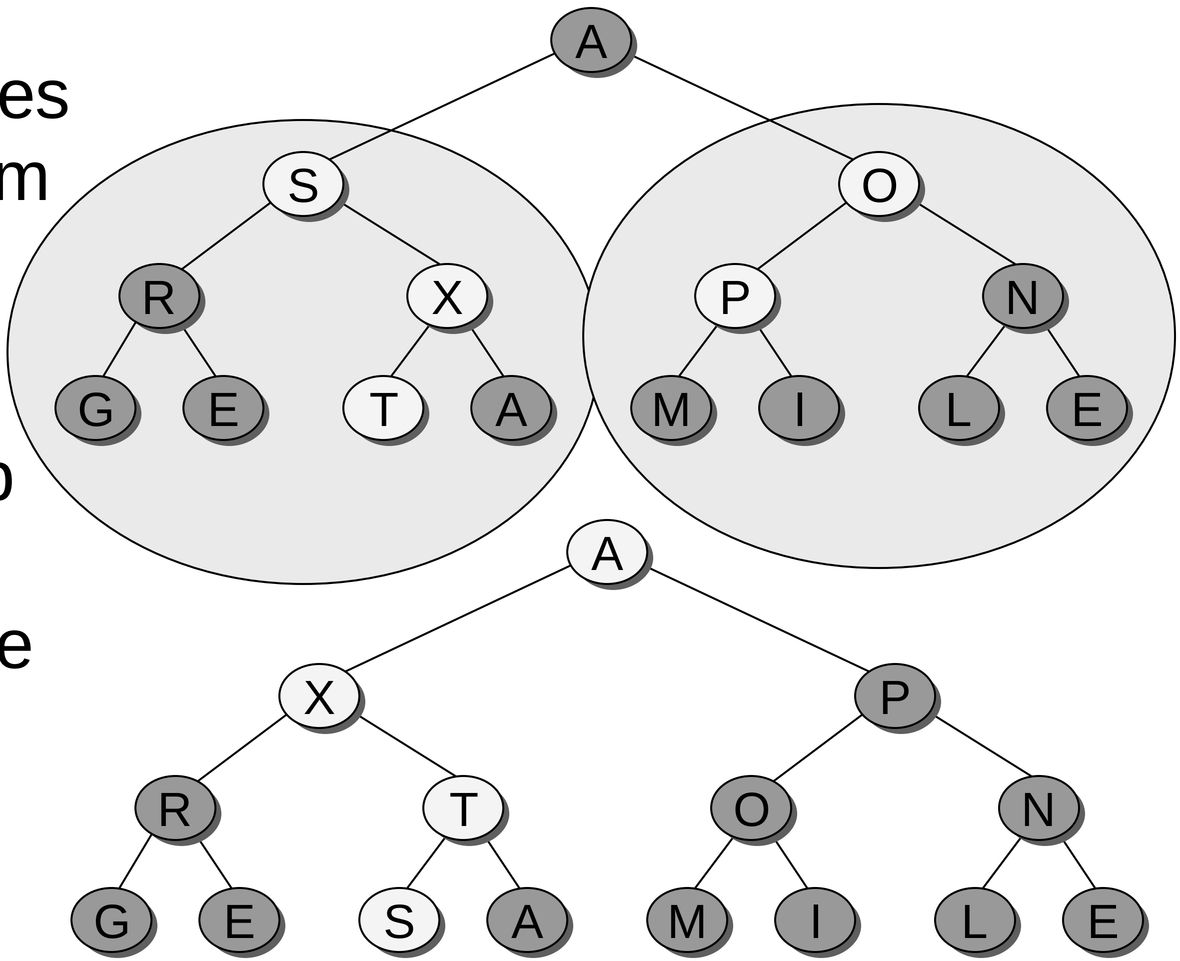
- Right to left
- Bottom-up




```
void heapsort(Item a[], int l, int r)
  { int k, N = r-l+1; Item* pq = a + l -1;
    for (k = N/2; k >= 1; k--)
      fixDown(pq, k, N);
    while (N > 1)
      { exch(pq[1], pq[N]);
        fixDown(pq, 1, --N); }
  }
```

Bottom-up Heap

- Note: most nodes are at the bottom
- Bottom up heap construction takes linear time



Radix Sort

- Decompose keys into pieces
  - Binary numbers are sequence of bytes
  - Strings are sequence of characters
  - Decimal number are sequence of digits
- Radix sort:
  - Sorting methods built on processing numbers one piece at a time
  - Treat keys as numbers represented in base R and work with individual digits
    - R = 10 in many applications where keys are 5- to 10-digit decimal numbers
    - Example: postal code, telephone numbers, SSN

Radix Sort

- If keys are integers
  - Can use R = 2, or a multiple of 2
- If keys are strings of characters
  - Can use R = 128 or 256 (aligns with a byte)
- Radix sort is based on the abstract operation
  - Extract the i^{th} digit from a key
- Two approaches to radix sort
  - Most-significant-digit (MSD) radix sorts (left-to-right)
  - Least-significant-digit (LSD) radix sorts (right-to-left)

Bits, Bytes and Word

- The key to understanding Radix sorts is to recognize that
  - Computers generally are built to process bits in groups called machine words (groups of bytes)
  - Sort keys are commonly organized as byte sequences
  - Small byte sequence can also serve as array indices or machine addresses
- Hence an abstraction can be used

Bits, Bytes and Word

- Def: A byte is a fixed-length sequence of bits; a string is a variable-length sequence of bytes; a word is a fixed-length sequence of bytes

```
#define bitsword 32
#define bitsbyte 8
#define bytesword 4
#define R (1 << bitsbyte)

#define digit(A, B)
  (((A) >> (bitsword-((B)+1)*bitsbyte)) & (R-1))

// Another possibility
#define digit(A, B) A[B]
```

- digit(A, 2)??

Binary Quicksort

- Partition a file based on leading bits
- Sort the sub-files recursively

```
quicksortB(int a[], int l, int r, int w)
  { int i = l, j = r;
    if (r <= l || w > bitsword) return;
    while (j != i)
      {
        while (digit(a[i], w) == 0 && (i < j)) i++;
        while (digit(a[j], w) == 1 && (j > i)) j--;
        exch(a[i], a[j]);
      }
    if (digit(a[r], w) == 0) j++;
    quicksortB(a, l, j-1, w+1);
    quicksortB(a, j, r, w+1);
  }
void sort(Item a[], int l, int r)
  {
    quicksortB(a, l, r, 0);
  }
```

Binary Quicksort

001	001	001	001
111	010	010	010
011	011	011	011
100	100	100	100
101	101	101	101
010	111	111	111

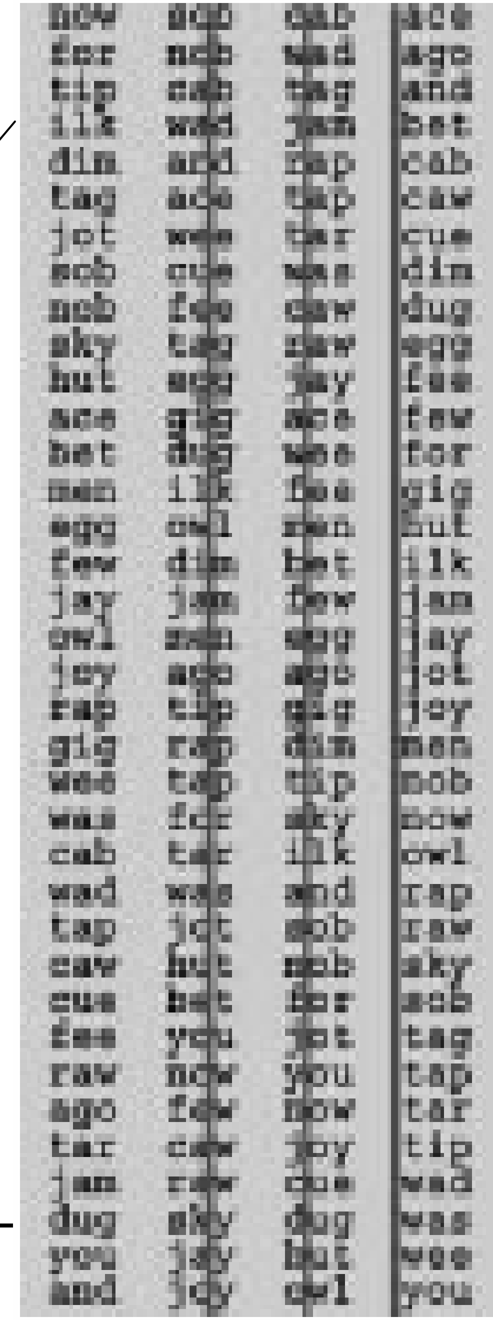
MSD Radix Sort

- Binary Quicksort is a MSD with R = 2;
- For general R, we will partition the array into R different bins/buckets

LSD Radix Sort

- Examine bytes
  Right to left

```
now sob cab ace
for nob wab ago
tip cab tag and
```



Symbol Table

- A symbol table is a data structure of items with keys that supports two basic operations: *insert* a new item, and *return* an item with a given key
  - Examples:
    - Account information in banks
    - Airline reservations

Symbol Table ADT

- Key operations
  - Insert a new item
  - Search for an item with a given key
  - Delete a specified item
  - Select the k^{th} smallest item
  - Sort the symbol table
  - Join two symbol tables

```
void STinit(int);
 int STcount();
void STinsert(Item);
Item STsearch(Key);
void STdelete(Item);
Item STselect(int);
void STsort(void (*visit)(Item));
```

Key-indexed ST

- Simplest search algorithm is based on storing items in an array, indexed by the keys

```
static Item *st;
static int M = maxKey;
void STinit(int maxN)
  { int i;
    st = malloc((M+1)*sizeof(Item));
    for (i = 0; i <= M; i++) st[i] = NULLitem;
  }
```

```
int STcount()
  { int i, N = 0;
    for (i = 0; i < M; i++)
      if (st[i] != NULLitem) N++;
    return N;
  }
void STinsert(Item item)
  { st[key(item)] = item; }
Item STsearch(Key v)
  { return st[v]; }
void STdelete(Item item)
  { st[key(item)] = NULLitem; }
Item STselect(int k)
  { int i;
    for (i = 0; i < M; i++)
      if (st[i] != NULLitem)
        if (k-- == 0) return st[i];
  }
void STsort(void (*visit)(Item))
  { int i;
    for (i = 0; i < M; i++)
      if (st[i] != NULLitem) visit(st[i]);
  }
```

Sequential Search based ST

- When a new item is inserted, we put it into the array by moving the larger elements over one position (as in insertion sort)
- To search for an element
  - Look through the array sequentially
  - If we encounter a key larger than the search key – we report an error

Binary Search

- Divide and conquer methodology
  - Divide the items into two parts
  - Determine which part the search key belongs to and concentrate on that part
    - Keep the items sorted
    - Use the indices to delimit the part searched.

```
Item search(int l, int r, Key v)
  { int m = (l+r)/2;
    if (l > r) return NULLitem;
    if eq(v, key(st[m])) return st[m];
    if (l == r) return NULLitem;
    if less(v, key(st[m]))
        return search(l, m-1, v);
    else return search(m+1, r, v);
  }
Item STsearch(Key v)
  { return search(0, N-1, v); }
```