Data Structures and Algorithms
Week 5

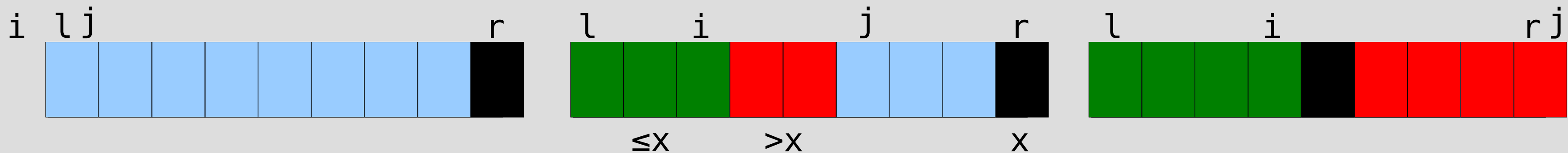
- Dynamic Data Structures
  - Pointers
  - Lists
  - Trees
- Abstract Data Types
  - Queue
  - Ordered Lists
  - Priority Queue

Previous Week

- Heapsort
  - Nearly complete binary trees
  - Heap data structure
  - Worst case: n log n
- Quicksort
  - a popular algorithm
  - very fast on average
  - worst case: n^2

Lomuto Partitioning

```
Partition(A,l,r)
  x := A[r];
  i := l-1;
  for j:=l to r-1 do
    if (A[j]<=x) then i++; swap A[i] and A[j]
  swap A[i+1] and A[r]
  return i+1
```



1. Identify worst cases for Lomuto partitioning
2. Propose fixes for the worst cases

Records

- Records are used to group a number of (different) fields.
- A *person* record may group *name, age, city, nationality, ssn*.
- The grouping of fields is a basic and often used technique.
- It is available in all programming languages.

Records in C

- In C a *struct* is used to group fields:

```
struct rec {
  int a;
  int b;
};

struct rec r;

int main() {
  r.a = 5; r.b = 8;
  printf(“The sum of a and b is %d\n”, r.a + r.b);
}

// gcc -o dummy dummy.c ; ./dummy
```

Records in Java

- In Java a *class* is used to group fields:

```
class rec { int a; int b; };

public class dummy {

  static rec r;

  public static void main(String args[]) {
    r = new rec();
    r.a = 15; r.b = 8;
    System.out.print("Adding a and b yields ");
    System.out.println(r.a + r.b);
  }
}
```

Recursive Data Structures/1

- The counterpart of recursive functions are recursively defined data structures.
- Example: list of integers

$$\text{list} = \begin{Bmatrix} \text{integer} \\ \text{integer}, \text{list} \end{Bmatrix}$$

- In C

```
struct list {
  int value;
  struct list tail;
};
```

Recursive Data Structures/2

- The storage space of recursive data structures is not known in advance.
  - It is determined by the number of elements that will be stored in the list.
  - This is only known during runtime (program execution).
  - The list can grow and shrink during program execution.

Recursive Data Structures/3

- There must be a mechanism to constrain the initial storage space of recursive data structures (it is potentially infinite).
- There must be a mechanism to grow and shrink the storage space of a recursive data structures during program execution.

Pointers/1

- A common technique is to allocate the storage space (memory) dynamically.
- That means the storage space is allocated when the program executes.
- The compiler only reserves space for an **address** to these dynamic parts.
- These addresses are called **pointers**.

Pointers/2

- integer i
- pointer p to an integer (55)
- record r with integer components a (17) and b (24)
- pointer s that points to r

Address	Variable	Memory
1af782	i	23
1af783	p	1af789
1af784	r	17
1af785		24
1af786	s	1af784
1af787		
1af788		
1af789		55
1af78a		

Pointers in C/1

1. To follow (chase, dereference) a pointer variable we write *p
   - `*p = 12`
2. To get the address of a variable i we write &i
   - `p = &i`
3. To allocate memory we use malloc(sizeof(Type))
   - `p = malloc(sizeof(int))`
4. To free storage space pointed to by a pointer p we use free
   - `free(p)`

Pointers in C/2

- To declare a pointer to type T we write T*
  - `int* p`
- Note that * is used for two purposes:
  - Declaring a pointer variable
    `int* p`
  - Following a pointer
    `*p = 15`
- In other languages these are syntactically different.

Pointers in C/3

- int i
  i = 23
- int* p
  p = malloc(sizeof(int))
  *p = 55
- struct rec r
  rec.a = 17
  rec.b = 24
- struct rec* s;
  s = &r

Address	Variable	Memory
1af782	i	23
1af783	p	1af789
1af784	r	17
1af785		24
1af786	s	1af784
1af787		
1af788		
1af789		55
1af78a		

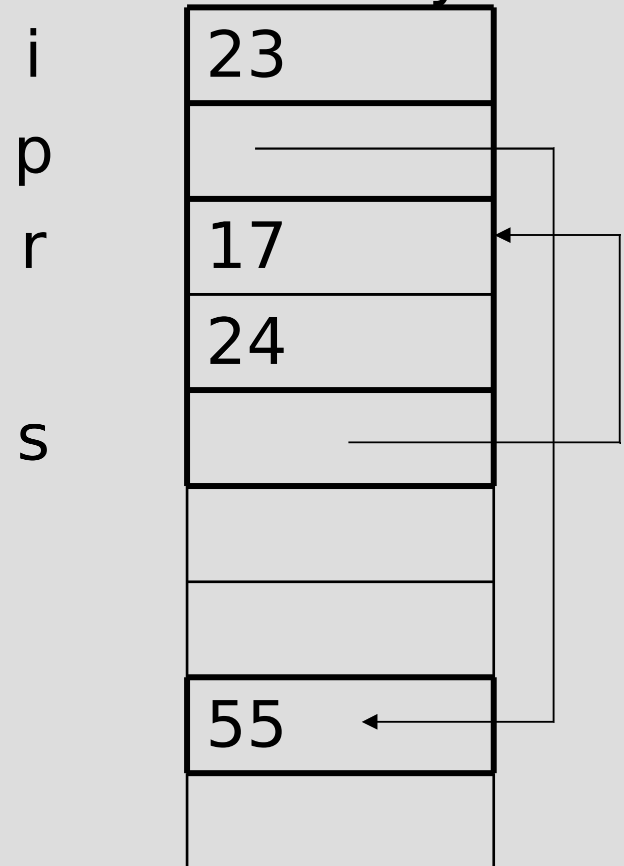
Pointers/3

- Pointers are only one mechanism to implement recursive data structures.
- The programmer does not have to be aware of their existence. The storage space can be managed automatically.
- In C the storage space has to be managed explicitly.
- In Java
  - an object is implemented as a pointer.
  - creation of objects (new) automatically allocates storage space.
  - accessing an object will automatically follow the pointer.
  - deallocation is done automatically (garbage collection).

Pointers/4

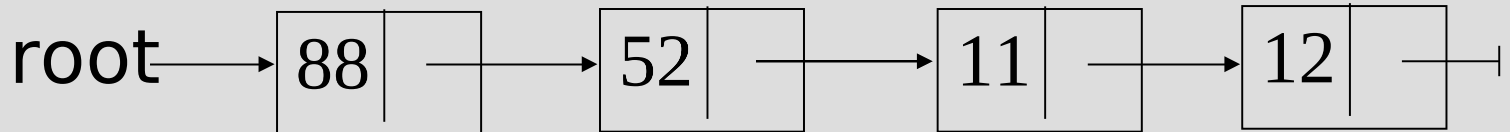
Alternative notation:

Address	Variable	Memory
1af782	i	23
1af783	p	1af789
1af784	r	17
1af785		24
1af786	s	1af784
1af787		
1af788		
1af789		55
1af78a		



Lists/1

- A list of integers:



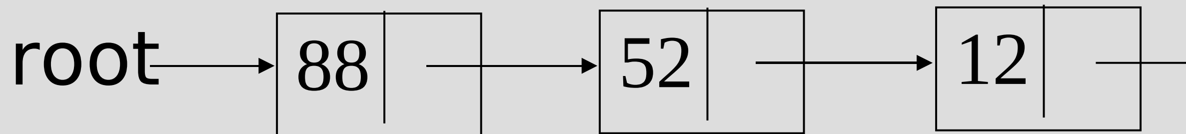
- Corresponding declaration in C:

```
struct node {
  int val;
  struct node* next;
}

struct node* root;
```

- Accessing a field: `(*p).a = p->a`

Lists/2

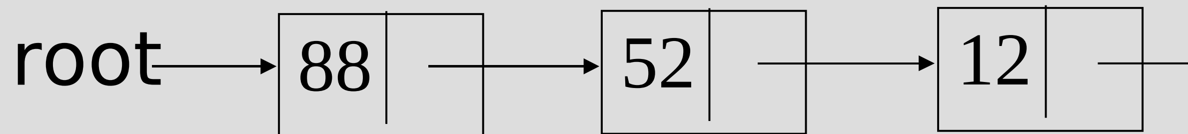
- Populating the list with integers:




```
root = malloc(sizeof(struct node));
root->val = 88;
root->next = malloc(sizeof(struct node));

p = root->next;
p.val = 52;
p->next = malloc(sizeof(struct node));

p = p->next;
p->val = 12;
p->next = NULL;
```

List Traversal

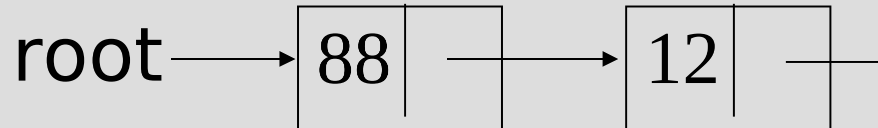
- Print all elements of a list:



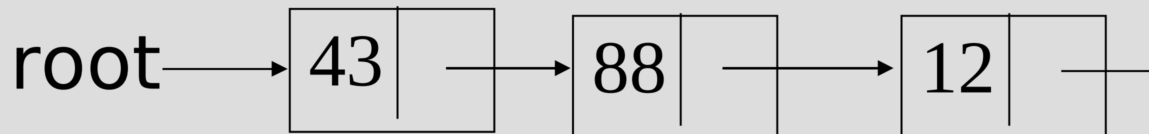

```
p = root;
while (p != null) {
  printf("%d,", p->val);
  p = p->next
}
printf("\n");
```

List Insertion/1

- Insert 43 at the beginning:




```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```



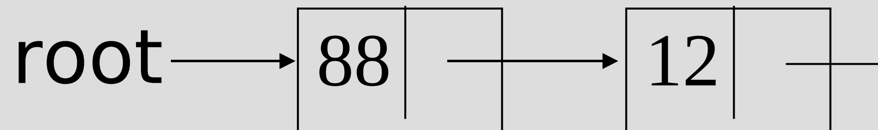
List Insertion/2

- Insert 43 at end: root


```
if (root == null) {
  root = malloc(sizeof(struct node));
  root->val = 43;
  root->next = null;
} else {
  q = root;
  while (q->next != null) { q = q->next; }
  q->next = malloc(sizeof(struct node));
  q->next->val = 43;
  q->next->next = null;
}
```

List Insertion/3

- Insert 43 at the end:



- Explain the following code fragment:

```
q = root;
while (q != null) { q = q->next; }
q = malloc(sizeof(struct node));
q->val = 43;
q->next = null;
```

List Deletion

- Delete element x from a non-empty list:

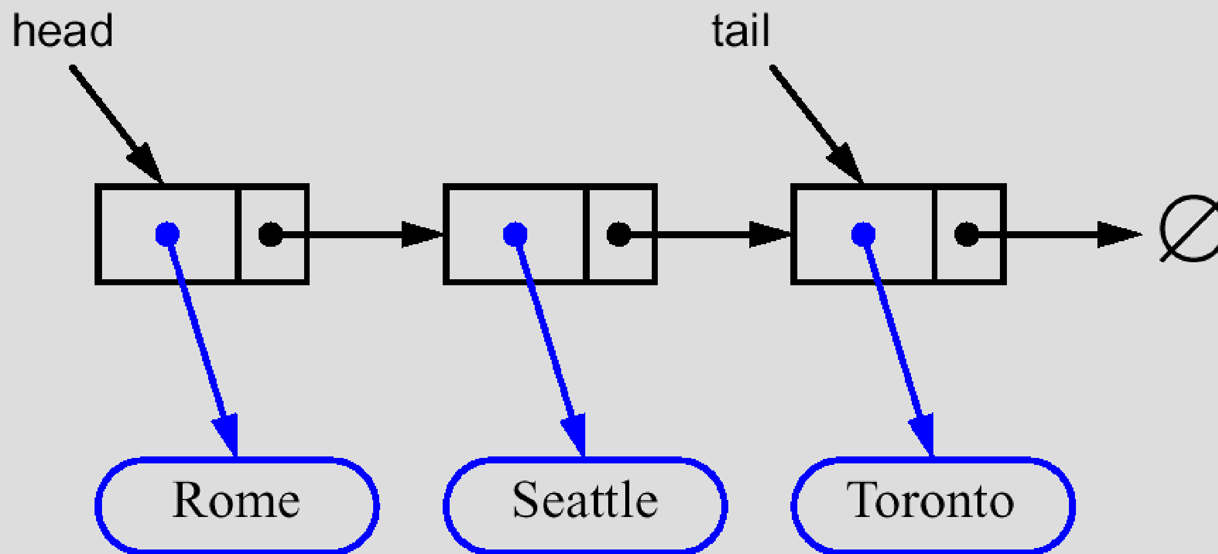
```
p = root;
if (p->val == x) {
  root = p->next;
  free(p);
} else {
  while (p->next != null && p->next->val != x) {
    p = p->next;
  }
  tmp = p->next;
  p->next = tmp->next
  free(tmp);
}
```

Performance of Lists

- Cost of operations:
  - Insertion at beginning: O(1)
  - Insert at end: O(n)
  - isEmpty: O(1)
  - Delete: O(n)
  - Print: O(n)

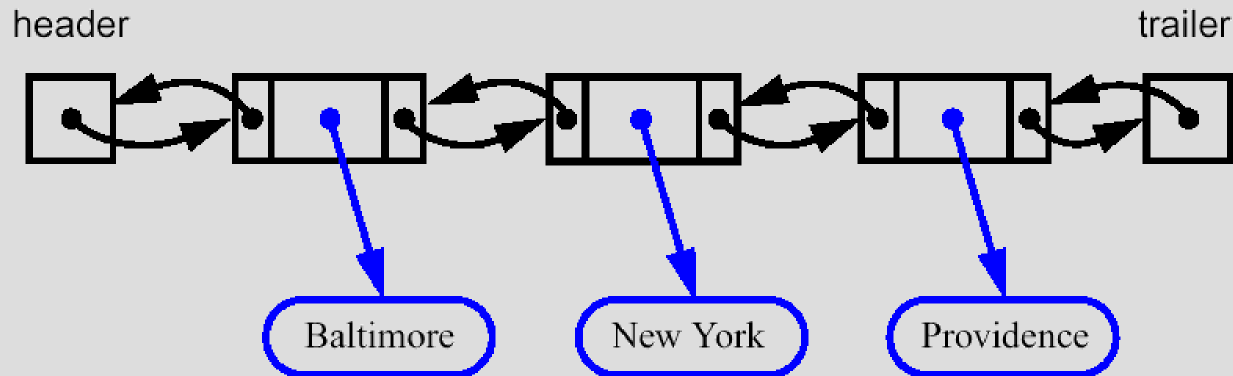
List with Explicit Head/Tail

- Instead of a *root* we can have a *head* and *tail*:

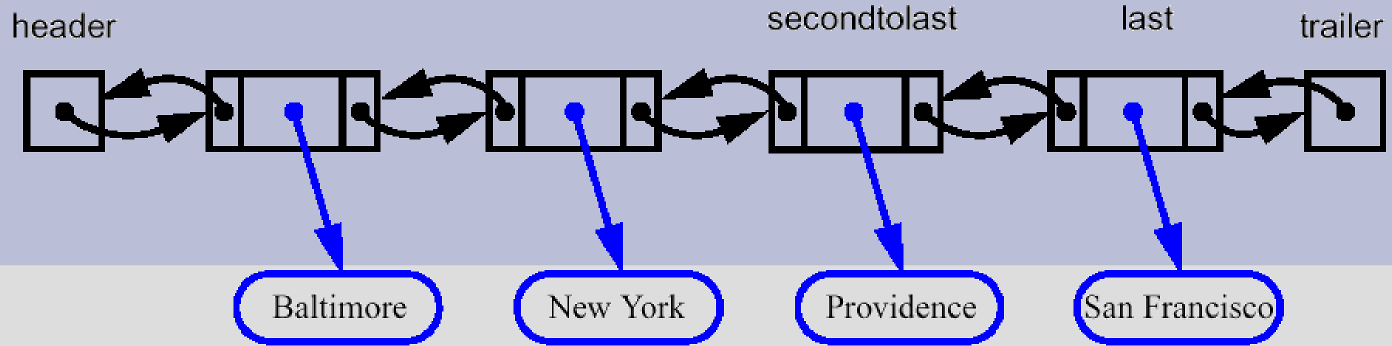


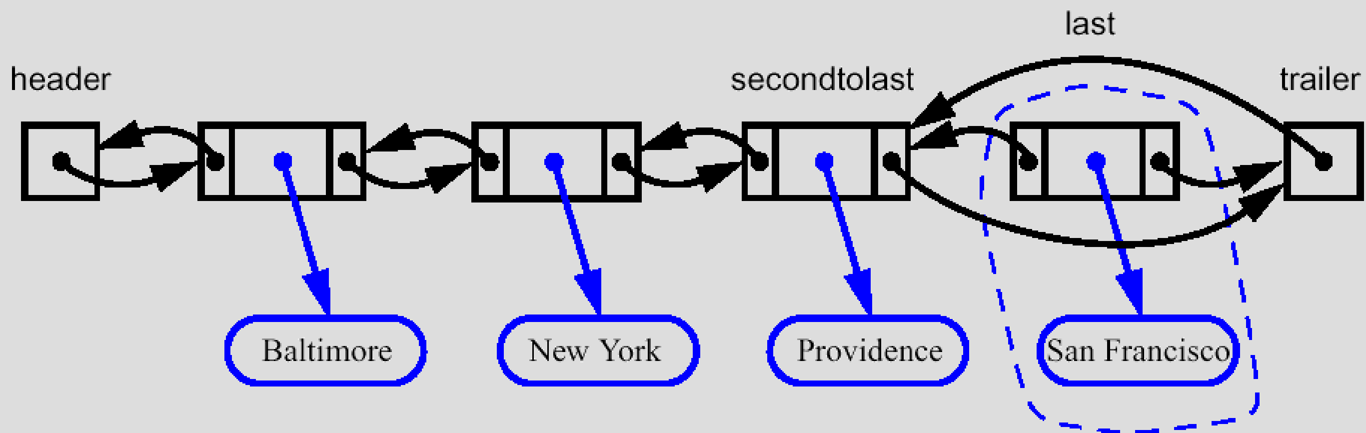
Doubly Linked Lists

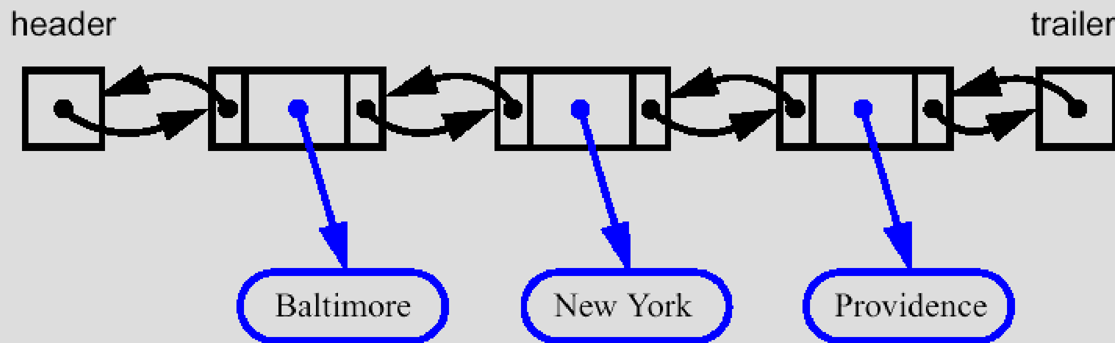
- To be able to quickly navigate back and forth in a list we use **doubly linked lists**.



- A node of a doubly linked list has a **next** and a **prev** link.


header
secondtolast
last
trailer
Baltimore
New York
Providence
San Francisco


last
header
secondtolast
trailer
Baltimore
New York
Providence
San Francisco


header
trailer
Baltimore
New York
Providence

Abstract Data Types (ADTs)

- An *ADT* is a mathematically specified entity that defines a set of its *instances*, with:
  - a specific *interface* – a collection of signatures of operations that can be invoked on an instance.
  - a set of *axioms* (*preconditions* and *post-conditions*) that define the semantics of the operations (i.e., what the operations do to instances of the ADT, but not how).

ADTs/2

- Why ADTs?
  - ADTs allows to break work into pieces that can be worked on independently – without compromising correctness.
    - They serve as *specifications* of *requirements* for the building blocks of solutions to algorithmic problems.
  - ADTs encapsulate *data structures* and algorithms that *implement* them.

ADTs/3

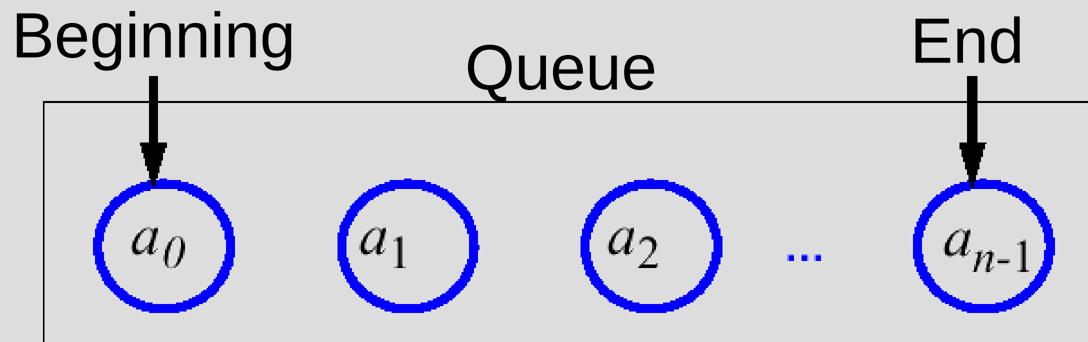
- Provides a language to talk on a higher level of abstraction.
- Allows to separate the concerns of *correctness* and the *performance analysis*
  1. Design the algorithm using an ADT
  2. Count how often different ADT operations are used
  3. Choose implementations of ADT operations
- ADT = Instance variables + procedures (Class = Instance variables + methods)

ADTs/4

- We discuss a number of popular ADTs:
  - Queues
  - Ordered Lists
  - Priority Queues
  - Trees
- They illustrate the use of lists and arrays.

Queues/1

- In a queue insertions and deletions follow the **first-in-first-out** (FIFO) principle.
- Thus, the element that has been in the queue for the longest time are deleted.
  - Example: Printer queue, …
- Solution: Elements are inserted at the **end** (enqueue) and removed from the **beginning** (dequeue).



Queues/2

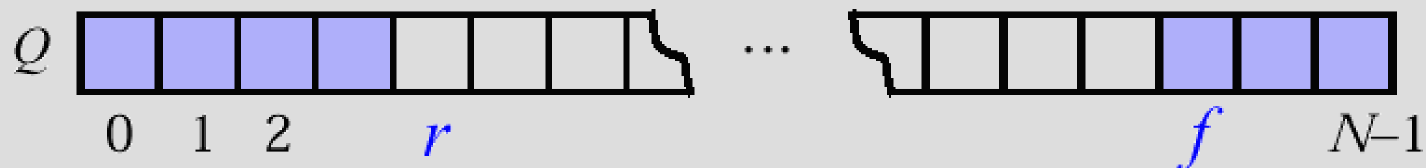
- Appropriate data structure:
  - Linked list, root: inefficient insertions
  - Linked list, head/tail: good
  - Array: fastest, limited in size
  - Doubly linked list: unnecessary
- What about inserting at the beginning and removing at the end?

An Array Implementation/1

- Create a queue using an array in a circular fashion
- A maximum size N is specified.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element (head, for dequeue)
  - r, index of the element after the last one (tail, for enqueue)

An Array Implementation/2

- "wrapped around" configuration



- what does $f = r$ mean?

An Array Implementation/3

15

```
int size()
   return (N-f+r) mod N
```

```
int isEmpty()
   return size() == 0
```

```
Element dequeue()
   if isEmpty() then Error
   x = Q[f]
   f = (f+1) mod N
   return x
```

```
void enqueue(Element x)
   if size()==N-1 then Error
   Q[r] = x
   r = (r+1) mod N
```

Ordered List/1

- In an ordered list elemenst are ordered according to a key.
- Example functions on ordered list:
  - `T* init()`
  - `bool isEmpty(T* l)`
  - `int first(T* l)`
  - `int last(T* l)`
  - `void print(T* l)`
  - `void insert(T* l, int x)`

Ordered List/2

- Declaration of an ordered list
  - Identical to unordered list
  - Operations are different

```
#define NULL 0

struct node {
  int val;
  struct node* next;
};

struct node* root;
```

Ordered List/3

- Initialization of an ordered list:

```
struct node* init() {
  return NULL;
}
```

- Retrieving the 1st element of an ordered list (-1 if the list is empty):

```
int first(struct node* l) {
if (l == NULL) return -1;
  else return l->val;
}
```

Ordered List/4

- Insertion into an ordered list:

```
void insert(struct node* l, int x) {
  struct node* p;
  struct node* q;

  if (root == NULL || root->val > x) {
    p = malloc(sizeof(struct node));
    p->val = x;
    p->next = root;
    root = p;
  } else {
…
```

Ordered List/5

- Insertion into an ordered list:

```
void insert(struct node* l, int x) {
  …
  } else {
    p = root;
    while (p->next != NULL && p->next->val < x)
      p = p->next;
    q = malloc(sizeof(struct node));
    q->val = x;
    q->next = p->next;
    p->next = q;
  }
}
```

Changing ADT instances/1

- Procedures implement operations on ADT instances.
- Some procedures must change the entry point (root, head, tail, etc).

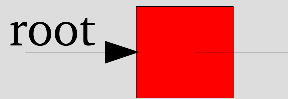
- Changing arguments passed to a procedure (e.g., root) is not effective.
- Accessing global variables is not good since this does not allow multiple instances.

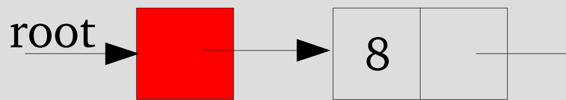
Changing ADT instances/2

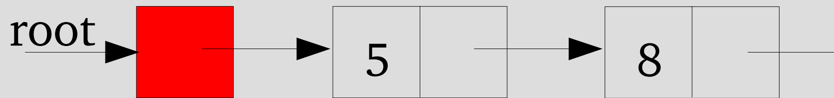
There are different ways to deal with ADT instances that might change:

A) Always use a **pointer to x** (sentinel) rather than x itself (this pointer never changes).
B) All operations that might have to change the entry point **return the new entry point**.
C) Include a **dummy element** so that the entry point (e.g., root) never changes.

Sentinel (Pointer to x)

Empty list: 

1 element list: 

2 element list: 

- Does not require a change of procedure calls.
- Extra space is independent of element size.

Code Sentinel/1

```
struct nd {
  int v;
  struct nd* n;
};

struct nd** lst1;

//-----------------------------------------------

struct nd** init() {
  struct nd **l;
  l = (struct nd**)malloc(sizeof(struct nd*));
  *l = NULL;
  return l;
}
```

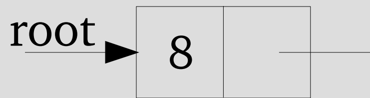
Code Sentinel/2

```
void insert(struct nd** l, int x) {
  struct nd* p;
  struct nd* q;
  if (*l == NULL || (*l)->v > x) {
    p = (struct nd*)malloc(sizeof(struct nd));
    p->v = x;
    p->n = *l;
    *l = p;
  } else {
    p = *l;
    while (p->n != NULL && p->n->v < x) { p = p->n; }
    q = (struct nd*)malloc(sizeof(struct nd));
    q->v = x;
    q->n = p->n;
    p->n = q;
```

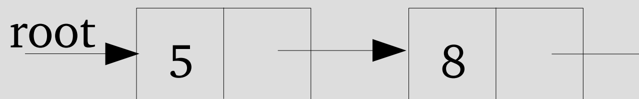
Return New Entry Point

Empty list: root

1 element list:



2 element list:



- Procedure calls change (handling of return values).
- Only one return value is possible.

Code Return New Entry Point/1

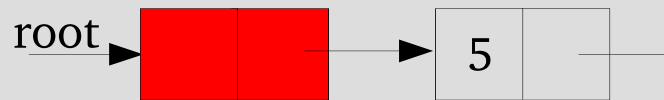
```
struct nd {
  int val;
  struct nd* next;
};

struct nd* lst1;

//-------------------------------------------------

struct nd* init() {
  return NULL;
}
```

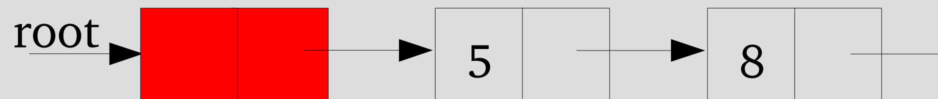
Code Return New Entry Point/2

```
struct nd*  insert(struct nd* l, int x) {
  struct nd* p;
  if (l == NULL || l->v >= x) {
    p = (struct nd*)malloc(sizeof(struct nd));
    p->v = x;
    p->n = l;
    return p;
  } else {
    l->n = insert(l->n, x);
    return l;
  }
}
```

Dummy Element

Empty list:

1 element list: 

2 element list: 

- Dummy element is sentinel with same structure as elements.
- Similarity helps to avoid special cases.
- Dummy element needs more space than sentinel.

Code Dummy Element/1

```
struct nd {
  int v;
  struct nd* n;
};

struct nd* lst1;

//----------------------------------------------------

struct nd* init() {
  struct nd* l;
  l = (struct nd*)malloc(sizeof(struct nd));
  l->n = NULL;
  return l;
}
```

Code Dummy Element/2

```
void insert(struct nd* l, int x) {
  struct nd* p;
  while (l->n != NULL && l->n->v < x) { l = l->n; }
  p = (struct nd*)malloc(sizeof(struct nd));
  p->v = x;
  p->n = l->n;
  l->n = p;
}
```

Priority Queues/1

- A priority queue is an *ADT* for maintaining a set S of elements, each with an associated value called key.
- A PQ supports the following operations
  - **HeapInsert**(S,x) insert element x in set S ($S := S \cup \{x\}$)
  - **HeapExtractMax**(S) returns and removes the element of S with the largest key

Priority Queues/2

- Removal of max takes constant time on top of Heapify Θ (log n)

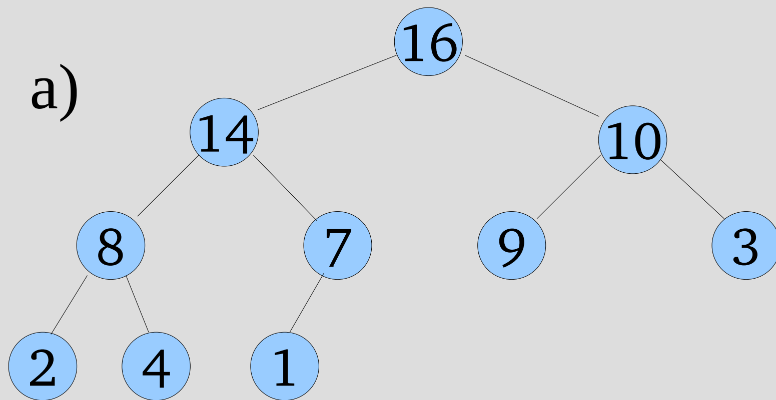
```
HeapExtractMax(A)
  // removes & returns largest elem of A
  max := A[1]
  A[1] := A[n]
  n := n-1
  Heapify(A, 1, n)
  return max
```

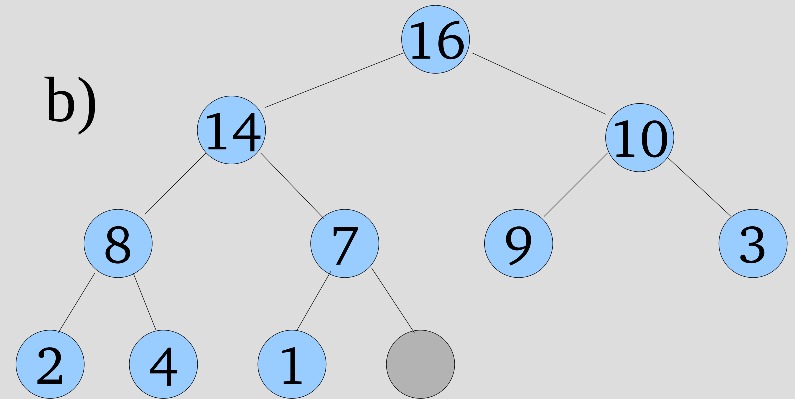
Priority Queues/3

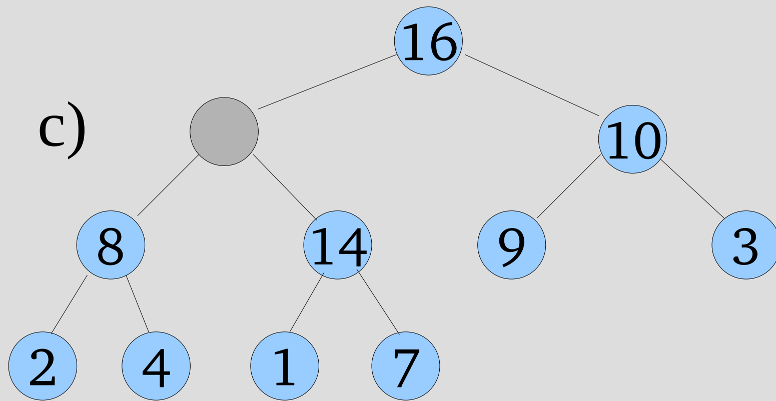
- Insertion of a new element
  - enlarge the PQ and propagate the new element from last place "up" the PQ
  - tree is of height *log n*, running time: Θ (log n)

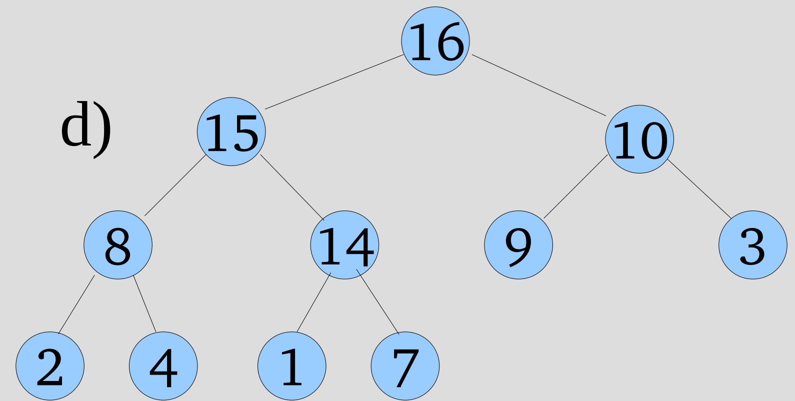
```
HeapInsert(A,key)
  n := n+1;
  i := n;
  while i > 1 and A[parent(i)] < key
    A[i] := A[parent(i)]
    i := parent(i)
  A[i] := key
```

Priority Queues/4









Priority Queues/5

- Applications:
  - job scheduling shared computing resources (Unix)
  - Event simulation
  - As a building block for other algorithms
- We used a heap and array to implement PQ. Other implementations are possible.

C, Java, Pointers/1

- Are objects in Java passed by reference or by value?
- What are possible values of Java variables?
- Is it possible to swap two objects passed as parameters to a Java method?

C, Java, Pointers/2

- What is the results of calling m?

```
void m(Point p) { p = NULL; }
```

- Are arrays in C passed by value or by reference?

C, Java, Pointers/3

- Show how to call the following procedure for multiplying two matrices such that it produces the wrong result:

```
void MM(int A[N][N], int B[N][N], int C[N][N]) {
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = 0;
      for (k=0; k<N; k++) {
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Summary

- Pointers
- Dynamic Data Structures
  - Lists (root, head/tail, doubly linked)
  - Trees
- Abstract Data Types
  - Type + Functions
  - Queue
  - Ordered Lists
  - Priority Queues

Next Week

- Binary Search Trees
- Red-Black Trees