Data Structures and Algorithms

Werner Nutt
Werner.Nutt@unibz.it
http://www.inf.unibz/it/~nutt

Chapter 5

Academic Year 2012-2013

Acknowledgements & Copyright Notice

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

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 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);
  }
}
```

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
```

Recursive Data Structures

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

$$\text{list} = \left\{ \begin{array}{c} \text{integer} \\ \text{integer}, \text{list} \end{array} \right\}$$

- 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

- 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. 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)), which returns an address in the memory heap
   - `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

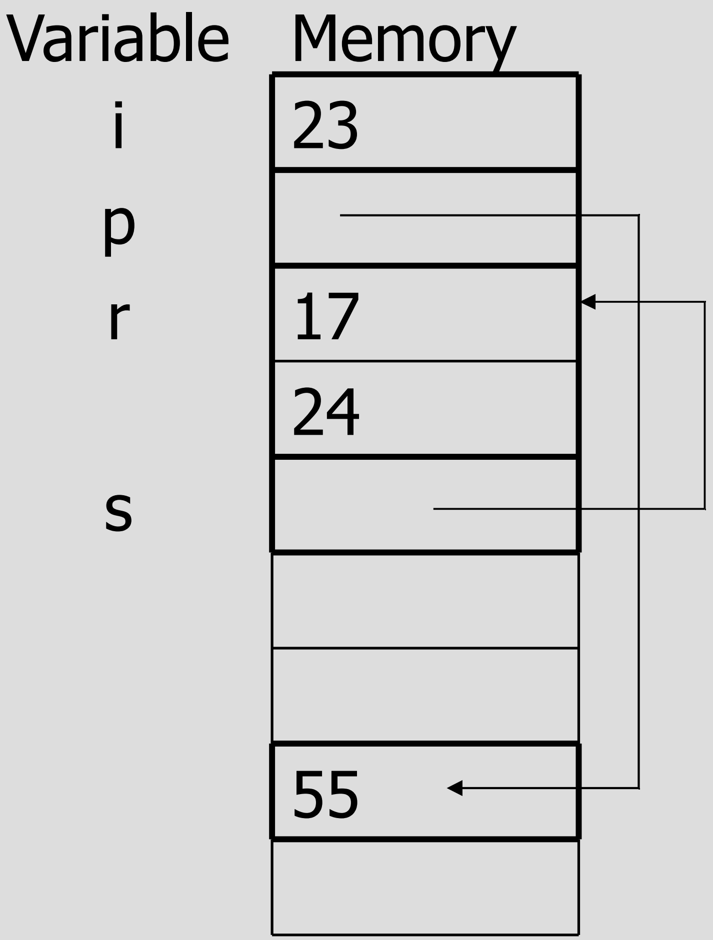
- int i
  i = 23
- int* p
  p = malloc(sizeof(int))
  *p = 55
- struct rec r
  rec.a = 17
  rec.b = 24
- struct rec* s;
  s = &r

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

Pointers in C/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		



Pointers/3

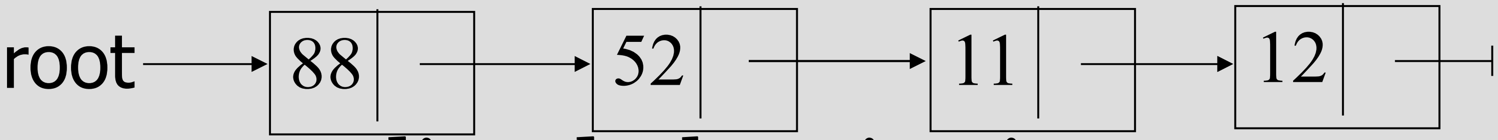
- Pointers are only one mechanism
  to implement recursive data structures.
- The programmer does not have to be aware
  of their existence.
  The storage space can be managed automatically.
- In C the storage space has to be managed explicitly.
- In Java
  - an object is implemented as a pointer.
 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

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

Lists

- A list of integers:



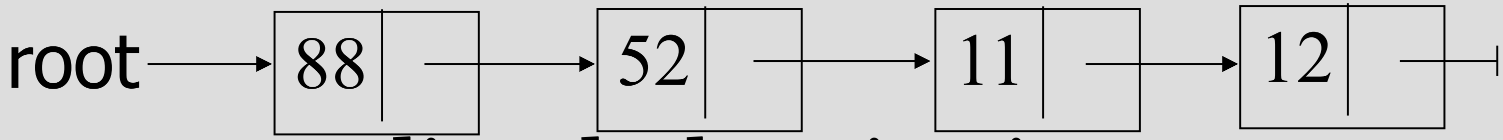
- Corresponding declaration in Java:

```
class node {
  int val;
  node next;
}

node root;
```

- Accessing a field: `p.a`

Lists/2

- A list of integers:



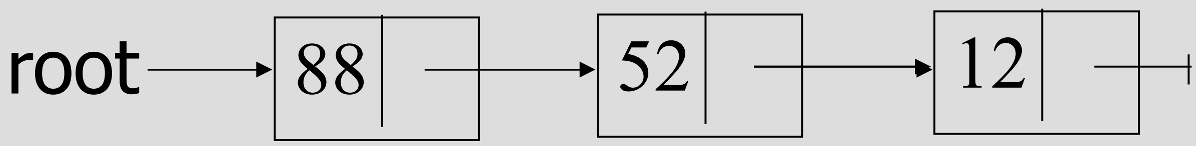
- Corresponding declaration in C:

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

struct node* root;
```

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

Lists/3

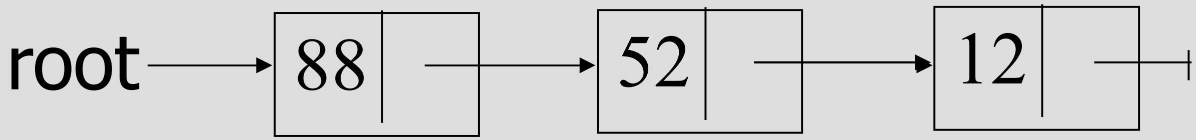
- Populating the list with integers (Java):




```
root = new node();

root.val = 88;
root.next = new node();

p = root.next;
p.val = 52;
p.next = new node();

p = p.next;
p.val = 12;
p.next = null;
```

Lists/4

- Populating the list with integers (C):

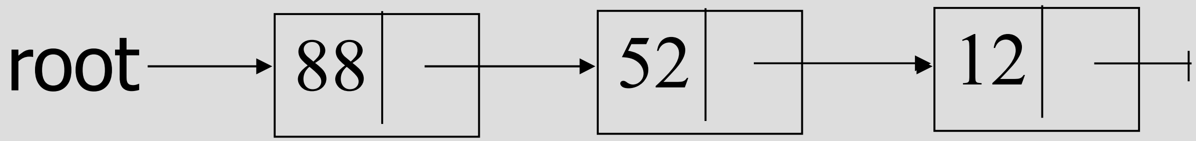



```
root = malloc(sizeof(struct node));
root->val = 88;
root->next = malloc(sizeof(struct node));

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

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

List Traversal

6
5

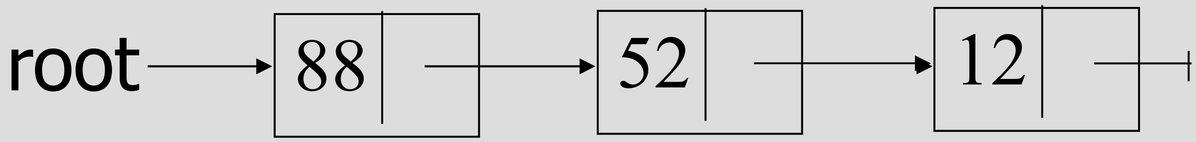
- Print all elements of a list (Java):




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

List Traversal

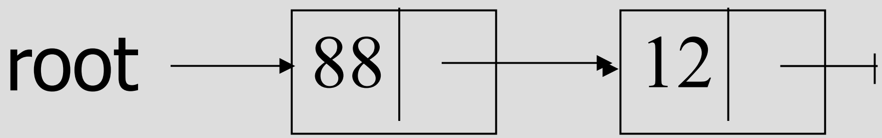
- Print all elements of a list (C):



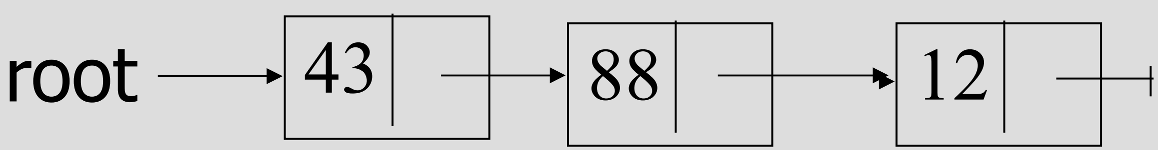

```
p = root;
while (p != null) {
  printf("%d,", p->val);
  p = p->next
}
printf("\n");
```

List Insertion

- Insert 43 at the beginning (Java):

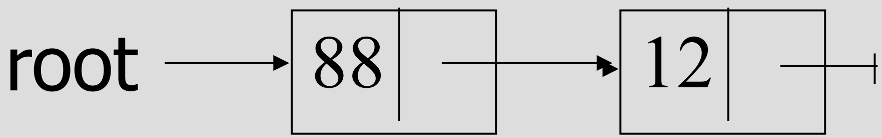



```
p = new node();
p.val = 43
p.next = root;
root = p;
```

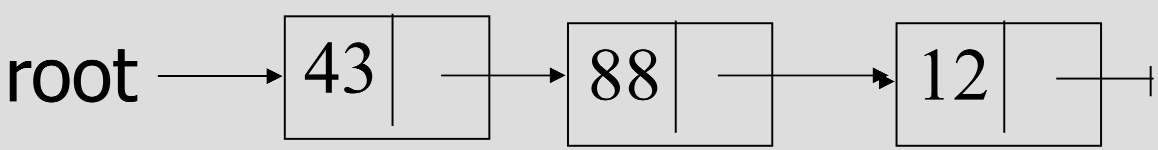


List Insertion

- Insert 43 at the beginning (C):

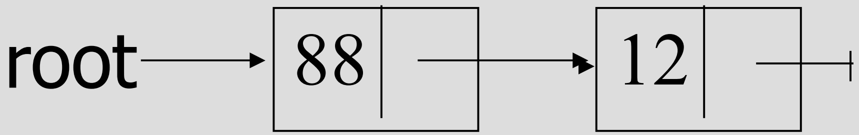



```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```



List Insertion/2

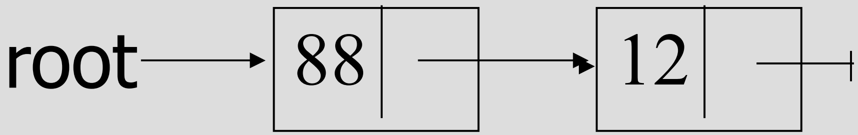
Insert 43 at end (Java):




```
if (root == null) {
  root = new node();
  root.val = 43;
  root.next = null;
} else {
  q = root;
  while (q.next != null) { q = q.next; }
  q.next = new node();
  q.next.val = 43;
  q.next.next = null;
}
```

List Insertion/2

Insert 43 at end (C):




```
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 Deletion

Delete element x from a non-empty list (Java):

```
p = root;
if (p.val == x) {
  root = p.next;
} // no need of freeing in java
else {
  while (p.next != null && p.next.val != x) {
    p = p.next;
  }
  tmp = p.next;
  p.next = tmp.next;
}
```

List Deletion

Delete element x from a non-empty list (C):

```
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);
}
```

List

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

Suggested exercises

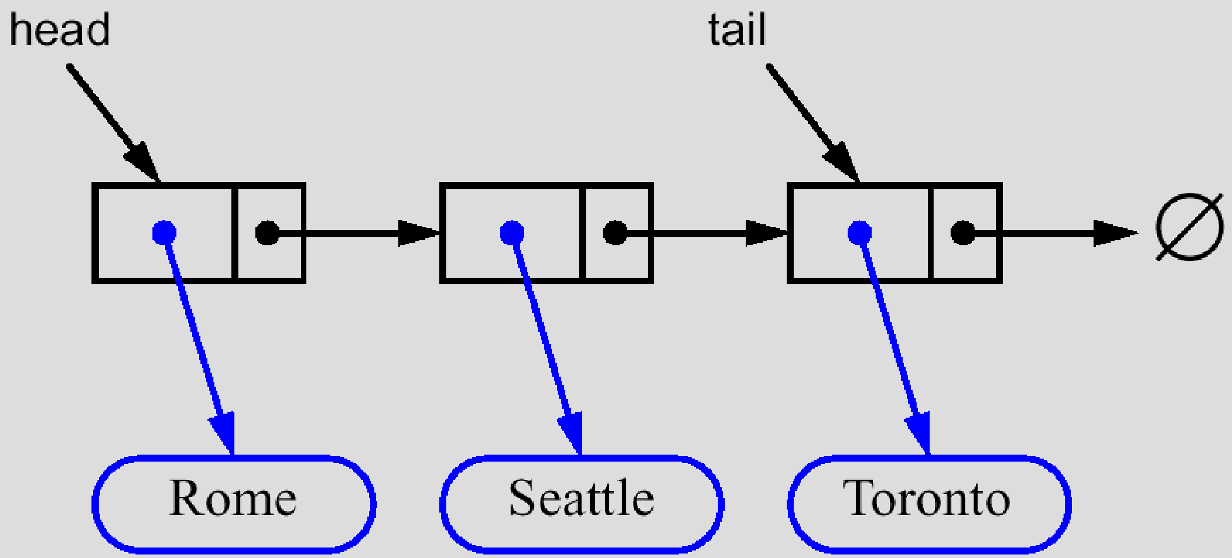
- Implement a linked list with the following functionalities: isEmpty, insertFirst, insertLast, search, deleteFirst, delete, print
- As before, with a recursive version of: insertLast, search, delete, print
  - are recursive versions simpler?
- Implement an efficient version of print which prints the list in reverse order

Variants of linked lists

- Linked lists with explicit head/tail
- Doubly linked lists

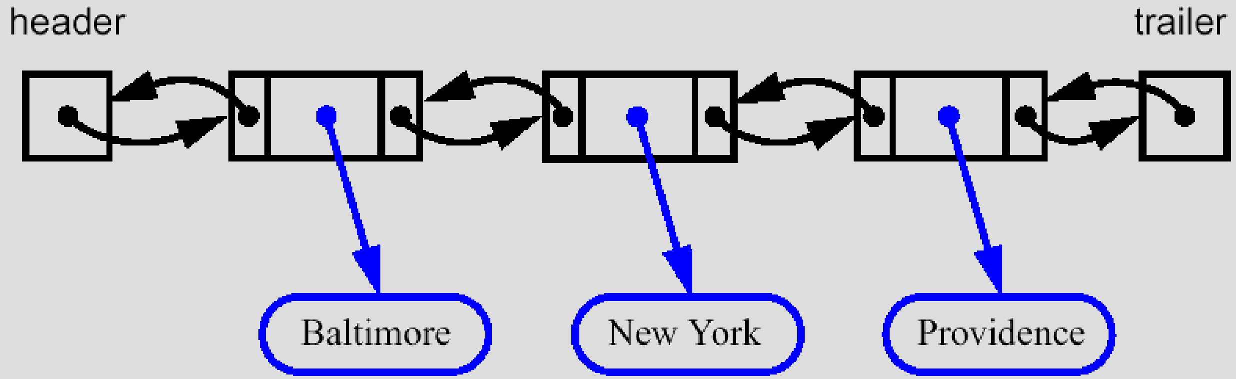
List with Explicit Head/Tail

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

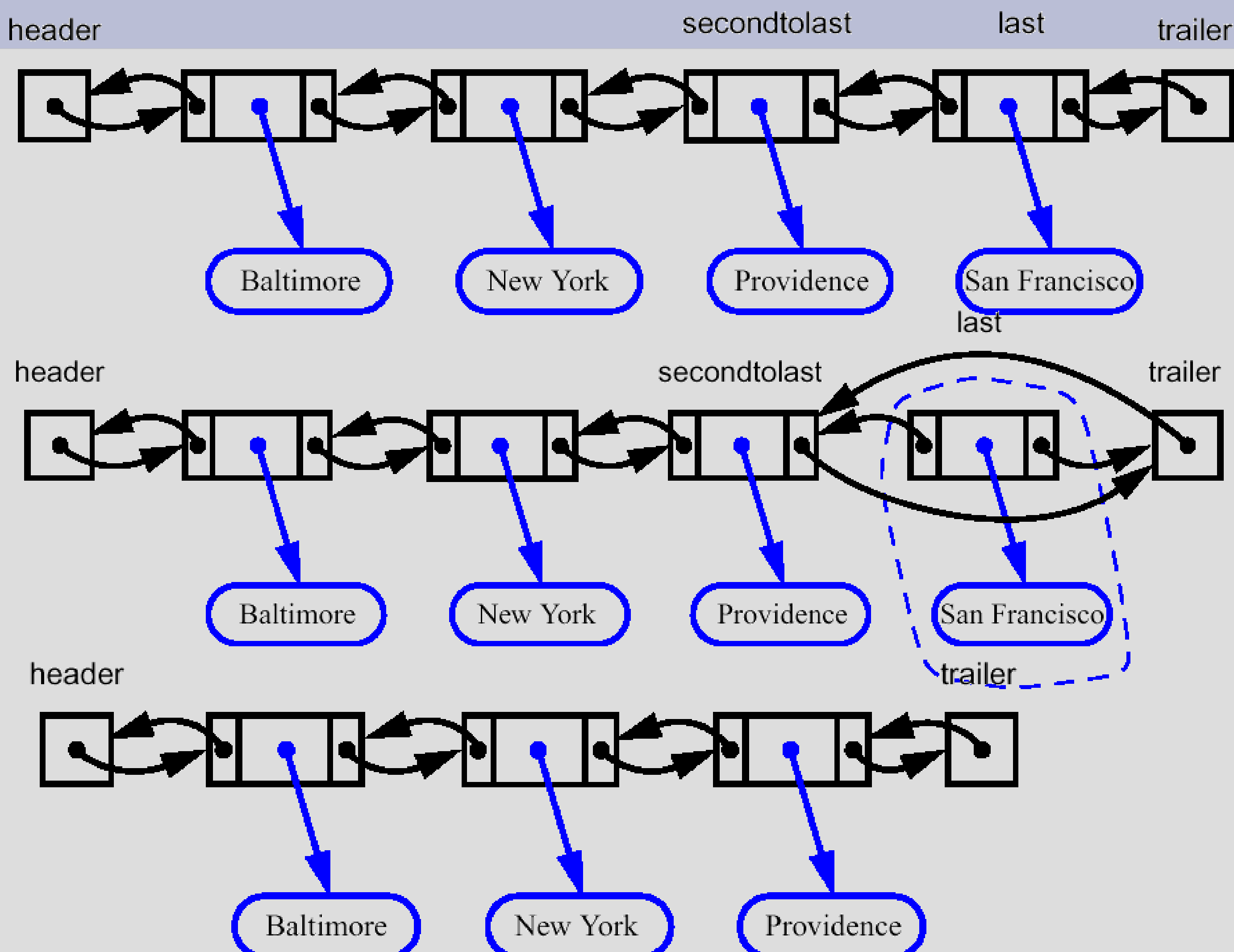


Doubly Linked Lists

- 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

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

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

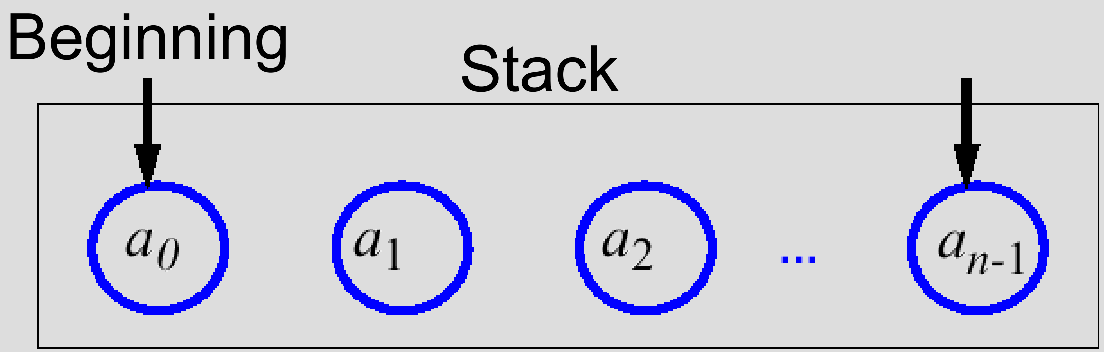
- We discuss a number of popular ADTs:
  - Stacks, Queues
  - Ordered Lists
  - Priority Queues
  - Trees (next chapter)
- They illustrate the use of lists and arrays.

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

Stacks

- In a stack, insertions and deletions follow the last**-in-first-out** (LIFO) principle.
- Thus, the element that has been in the queue for the shortest time are deleted.
  - Example: OS stack, ...
- Solution: Elements are inserted at the **beginning** (push) and removed from the **beginning** (pop).



Stacks/2

- Appropriate data structure:
  - Linked list, one root: good
  - Array: fastest, limited in size
  - Doubly linked list: unnecessary

An Array Implementation

- Create a stack using an array
- A maximum size *N* is specified.
- The stack consists of an N-element array S and one integer variable *count*:
  - *count:* index of the front element (head)
  - *count* represents the position where to insert next element, and the number of elements in the stack

An Array Implementation/2

```
int size()
   return count
```

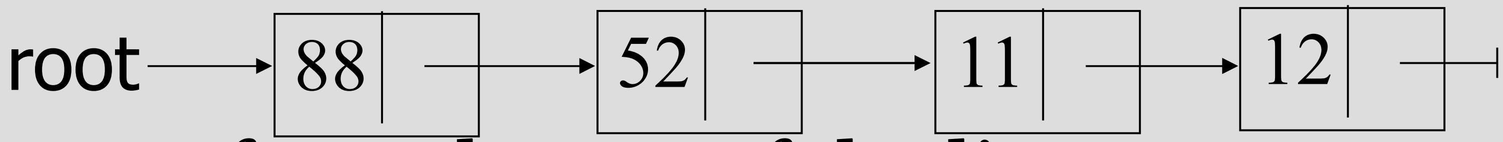
```
int isEmpty()
   return (count == 0)
```

```
Element pop()
   if isEmpty() then Error
   x = S[count-1]
   count--;
   return x
```

```
void push(element x)
   if count==N then Error;
   S[count] = x;
   count++;
```

A Linked-List implementation

- A list of integers:



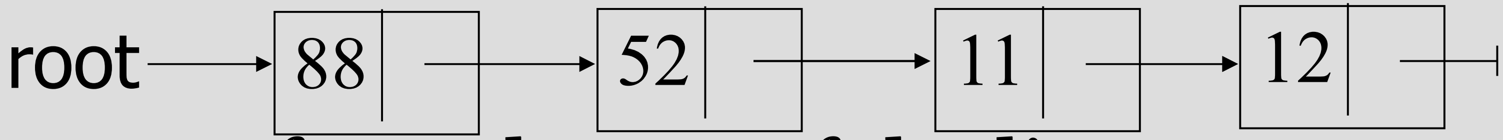
- Insert from the top of the list

```
push(element x):

node p = new node();
p.val = x;
p.next = root;
root = p;
```

- Constant-time operation!

A Linked-List implementation

- A list of integers:



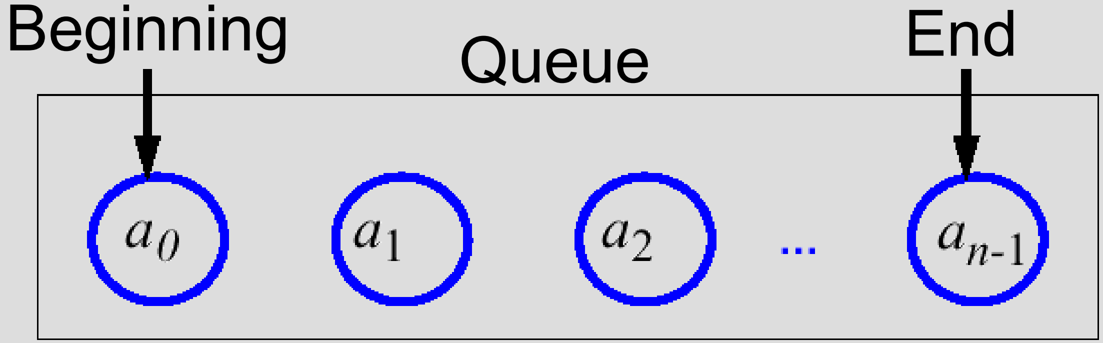
- Extract from the top of the list

```
Element pop():

x = root.val;
root = root.next;
Return x;
```

- Constant-time operation!

Queues

- 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

An Array Implementation

- 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

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

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

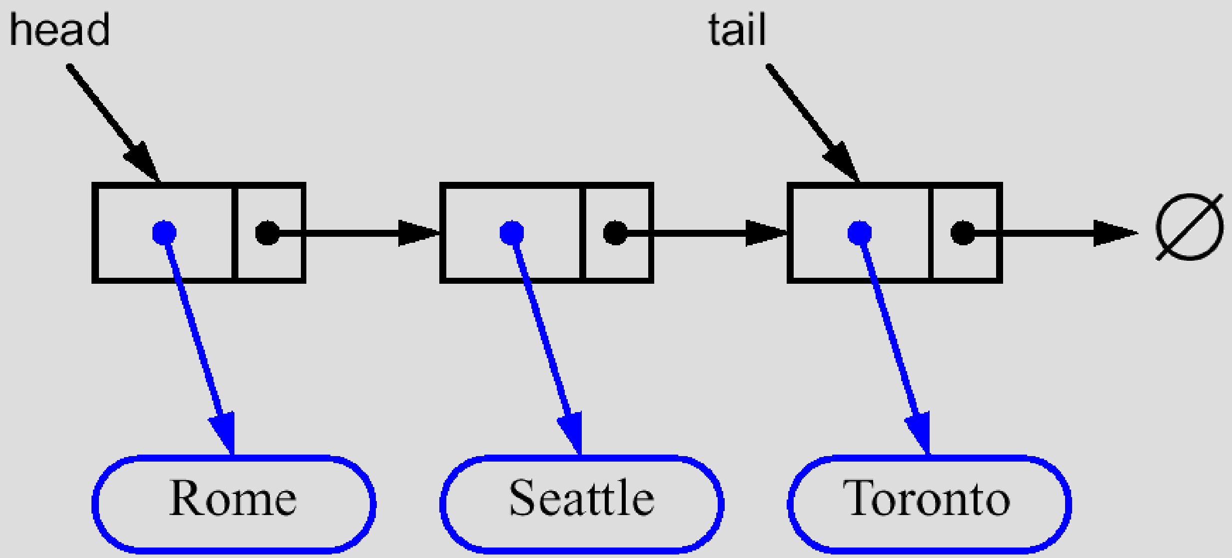
```
Element dequeue()
   if isEmpty() then Error
   x = Q[f]
   f = (f+1) mod N
   return x
```

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

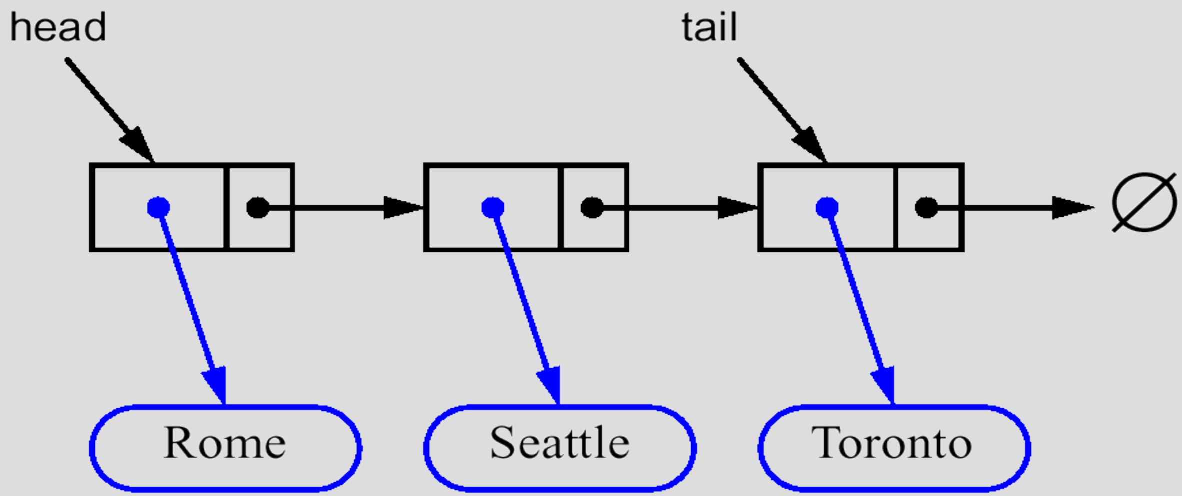
We assume arrays as in Java, with indexes from 0 to n-1

A Linked-List Implementation

- Use linked-list with head and tail
- Insert in tail, extract from head



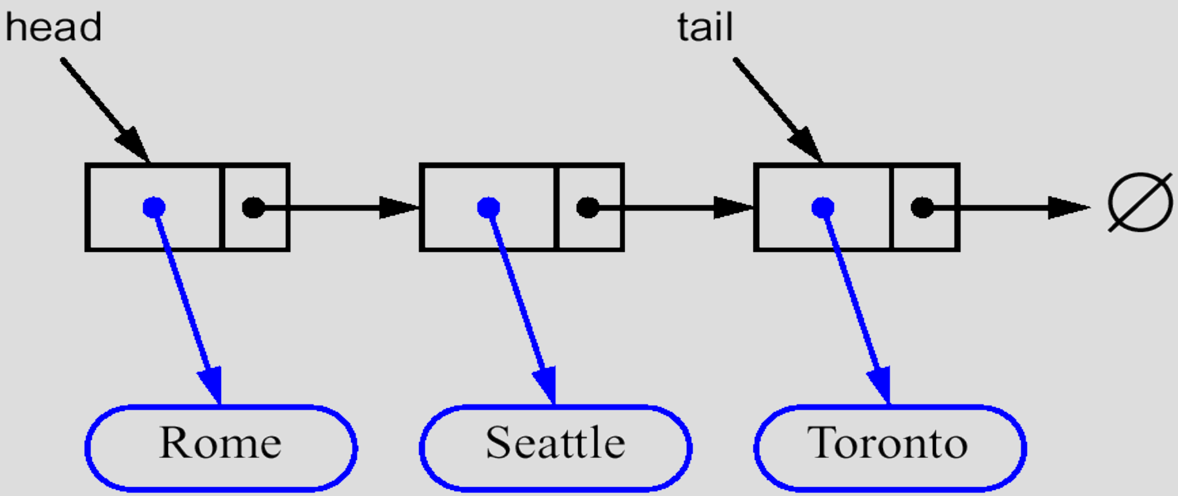
A Linked-List implementation/2



- Insert at the end of the list: O(1)

```
enqueue(element x):
node p = new node();
p.info = x; p.next = null;
tail.next=p;
tail=tail.next;
```

A Linked-List implementation/3



- Extract from the top of the list: O(1)

```
Element dequeue():
x = root.info;
root = root.next;
Return x;
```

Suggested exercises

- Implement stack and queue as arrays
- Implement stack and queue as linked lists, with the same interface as the array implementation

Suggested exercises/2

- Suppose a queue of integers is implemented with an array of 8 elements: draw the outputs and status of such array after the following operations:
  - enqueue 2,4,3,1,7,6,9
  - dequeue 3 times
  - Enqueue 2,3,4

  Can we enqueue any more element?
- Try the same with a stack
- Try similar examples (also with a stack)

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Queue
  - Ordered Lists
  - Priority Queue

Ordered List

- In an ordered list elements are ordered according to a key.
- Example functions on ordered list:
  - `init()`
  - `isEmpty()`
  - `Search(element x)`
  - `delete(element x)`
  - `print()`
  - `insert(element x)`

Ordered List/2

- Declaration of an ordered list identical to unordered list
- Some operations (search, and hence insert and delete) are slightly different

Ordered List/3

- Insertion into an ordered list (java):

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

  if (root == NULL || root.val > x) {
    p = new node();
    p.val = x;
    p.next = root;
    root = p;
  } else {
…
```

Ordered List/4

- Insertion into an ordered list (java):

```
void insert(node l, int x) {
  ...
  } else {
    p = root;
    while (p.next != NULL && p.next.val < x)
      p = p.next;
    q = new node();
    q.val = x;
    q.next = p.next;
    p.next = q;
  }
}
```

Ordered List/5

- Insertion into an ordered list (C):

```
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/6

- Insertion into an ordered list (C):

```
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;
  }
}
```

Ordered List

- Cost of operations:
  - Insertion: O(n)
  - Check isEmpty: O(1)
  - Search: O(n)
  - Delete: O(n)
  - Print: O(n)

Suggested exercises

- Implement an ordered list with the following functionalities: isEmpty, insert search, delete, print
- Implement also deleteAllOccurrences()
- As before, with a recursive version of: insert, search, delete, print
  - are recursive versions simpler?
- Implement an efficient version of print which prints the list in reverse order

Data Structures and Algorithms
Chapter 5

- Dynamic Data Structures
  - Records, Pointers
  - Lists
- Abstract Data Types
  - Stack, Queue
  - Ordered Lists
  - Priority Queue

Priority Queues

- 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
  - **Insert**(S,x) insert element x in set S ($S := S \cup \{x\}$)
  - **ExtractMax**(S) returns and removes the element of S with the largest key
- One way of implementing it: a heap

Priority Queues/2

- Removal of max takes constant time on top of Heapify $\Theta(log\ n)$

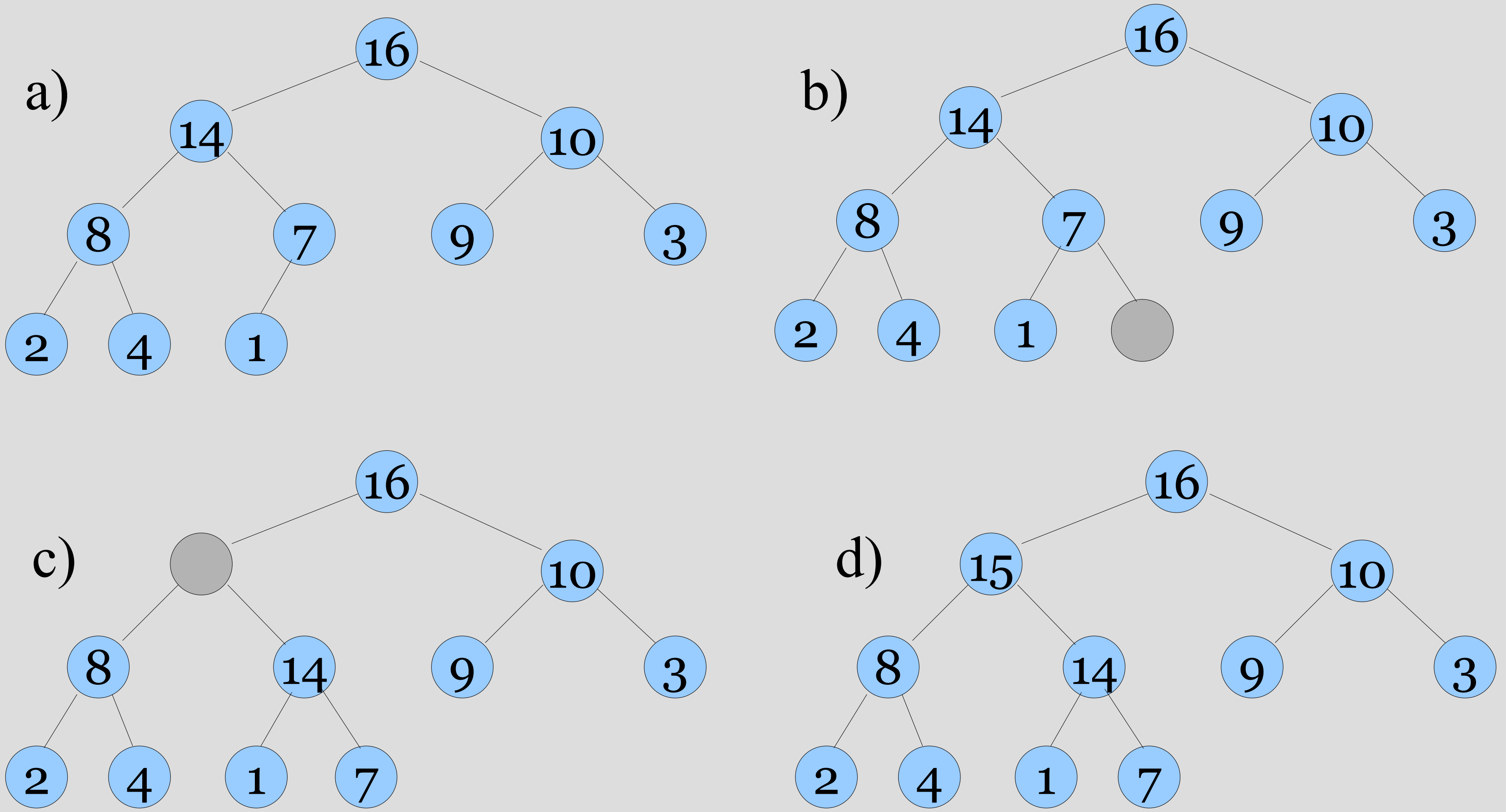
```
ExtractMax(A)
  // removes & returns largest elem of A
  max := A[1]
  A[1] := A[n]
  n := n-1
  Heapify(A, 1, n)
  return max
```

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)

```
Insert(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 an array to implement PQ. Other implementations are possible.

Suggested exercises

- Implement a priority queue
- Consider the PQ of previous slides. Draw the status of the PQ after each of the following operations:
    - Insert 17,18,18,19
    - Extract four numbers
    - Insert again 17,18,18,19
- Build a PQ from scratch, adding and inserting elements at will, and draw the status of the PQ after each operation

Summary

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

Next Chapter

- Binary Search Trees
- Red-Black Trees