Data Structures and Algorithms

Chapter 5

Dynamic Data Structures and Abstract Data Types

Werner Nutt

Acknowledgments

- The course follows the book “Introduction to Algorithms‘”, by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for this course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Chapter 5: Overview

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

DSA, Chapter 5: Overview

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

DSA, Chapter 5: Overview

- 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*
- 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

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

- In Java:

```
class List{
    int value;
    List tail;
};
```

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 mechanisms

- to constrain the initial storage space
  of recursive data structures
  (it is potentially infinite)
- 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

1

- 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

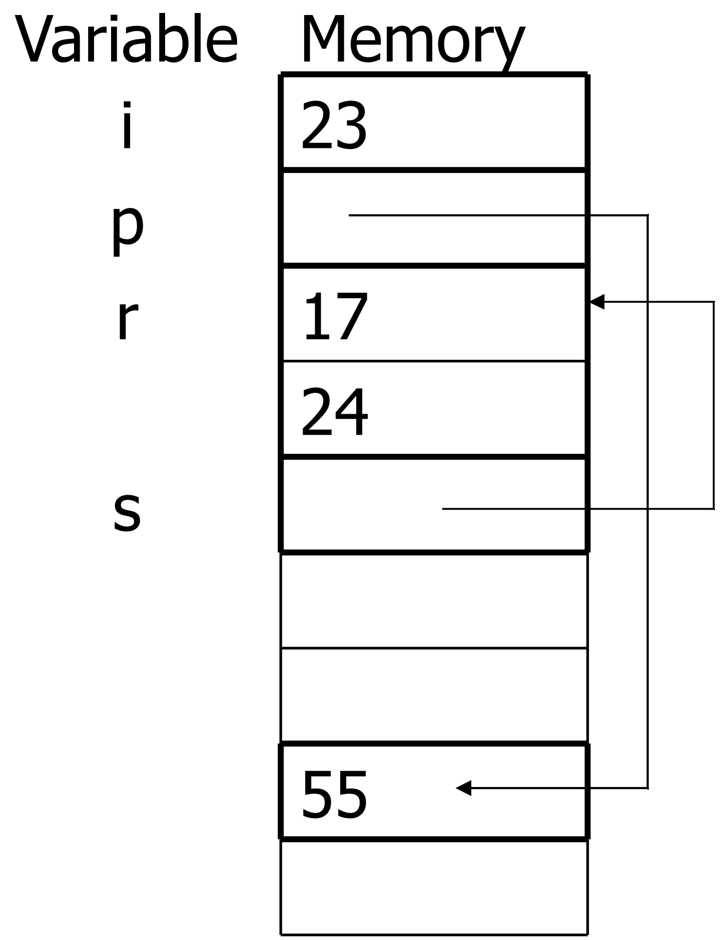
- int i
  i = 23
- int* p
  p = malloc(sizeof(int))
  *p = 55
- struct rec r
  r.a = 17
  r.b = 24
- struct rec* s;
  s = &r

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

Alternate 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

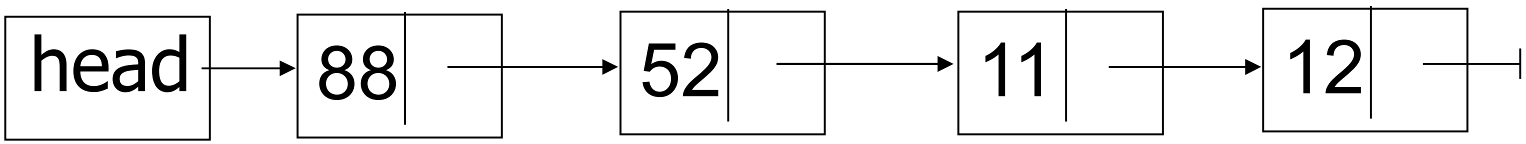
- Pointers are only one mechanism
  to implement recursive data structures
- Programmers need not be aware of their existence
  The storage space can be managed automatically
- In C the storage space has to be managed explicitly
- In Java
  - an object is implemented as a pointer
  - creation of objects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

DSA, Chapter 5: Overview

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

Lists

- A list of integers:



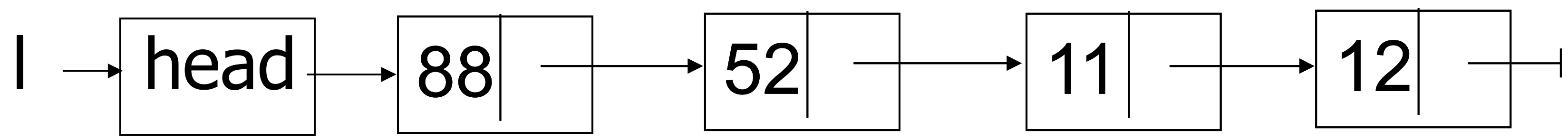
- Corresponding declaration in Java:

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

class List {
   Node head;
}
```

- Accessing a field: `p.a`

Lists/3

- Populating the list with integers (Java):

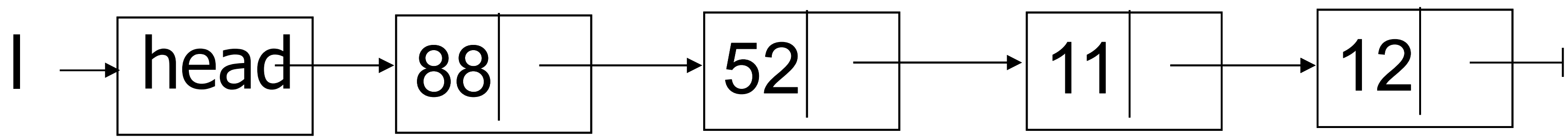



```
l = new List();
l.head = new Node();
l.head.val = 88;
l.head.next = new Node();

p = l.head.next;
p.val = 52;
p.next = new Node();
```

```
p = p.next;
p.val = 11;
p.next = new Node();

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

List Traversal

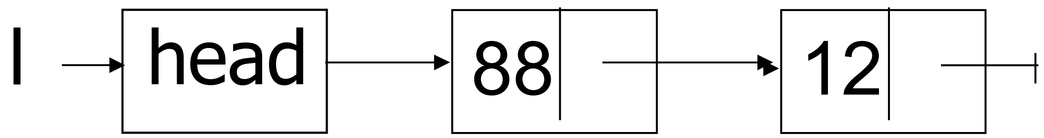
- Print all elements of a list (Java):



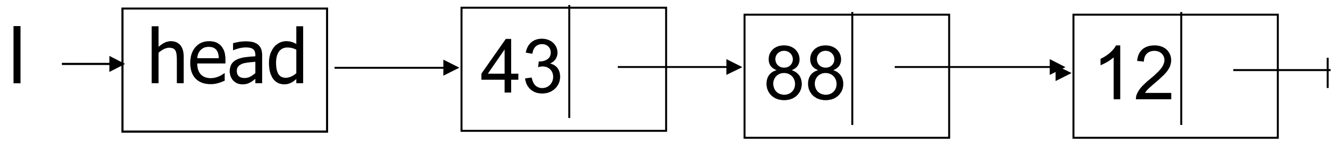

```
p = l.head;
while (p != null) {
   System.out.printf(“%d,”, p.val);
   p = p.next
}
System.out.printf(“\n”);
```

List Insertion

- Insert 43 at the beginning (Java):

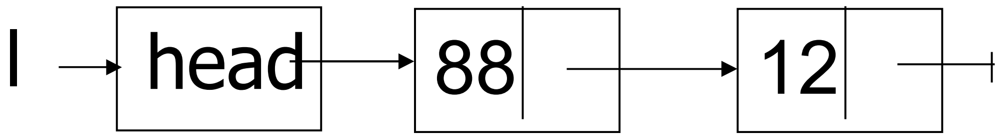



```
n = new Node();
n.val = 43
n.next = l.head;
l.head = n;
```



List Insertion/2

- Insert 43 at end (Java):




```
n = new Node();
n.val = 43;
n.next = null;
if (head == null) {
  head = n;
} else {
  p = head;
  while (p.next != null) { p = p.next; }
  p.next = n;
}
```

List Deletion

- Delete (first) node with value v from a non-empty list (Java):

```
p = l.head;
if (p.val == v) {
  head = p.next;
}
else {
  while (p.next != null && p.next.val != v)
  {
    p = p.next;
  }
  if (p.next != null){
      p.next = p.next.next;
}
```

Lists

Cost of operations:

- insert 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

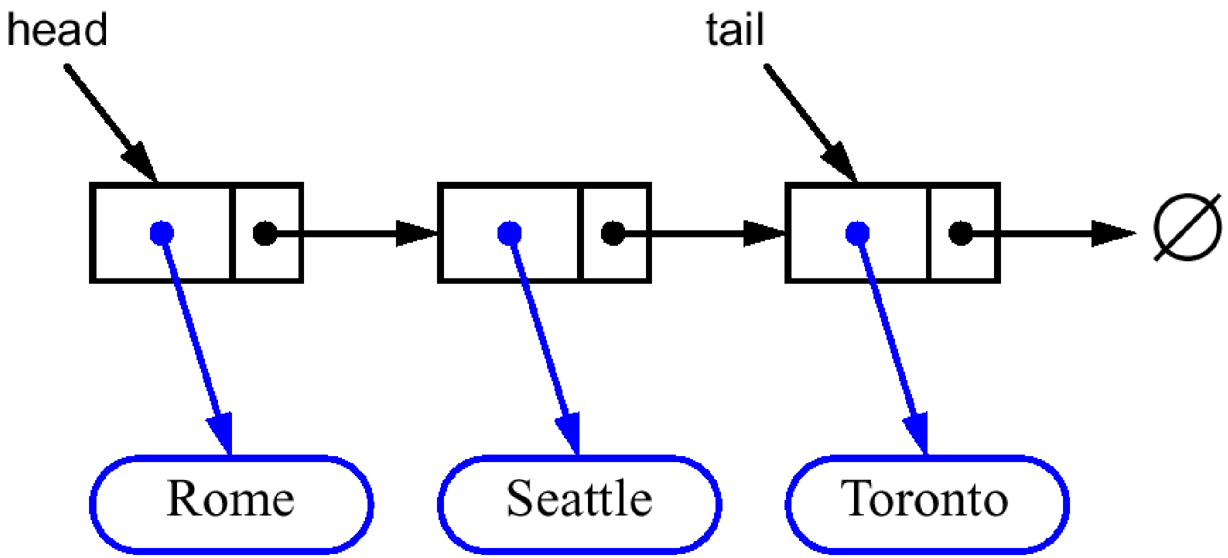
- Implement a linked list with the following functionalities: isEmpty, insertFirst, insertLast, search, deleteFirst, delete, print
- As before, with a recursive version of: insertLast, search, delete, print
  - are recursive versions simpler?
- Implement an efficient version of print which prints the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

Variants of Linked Lists

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

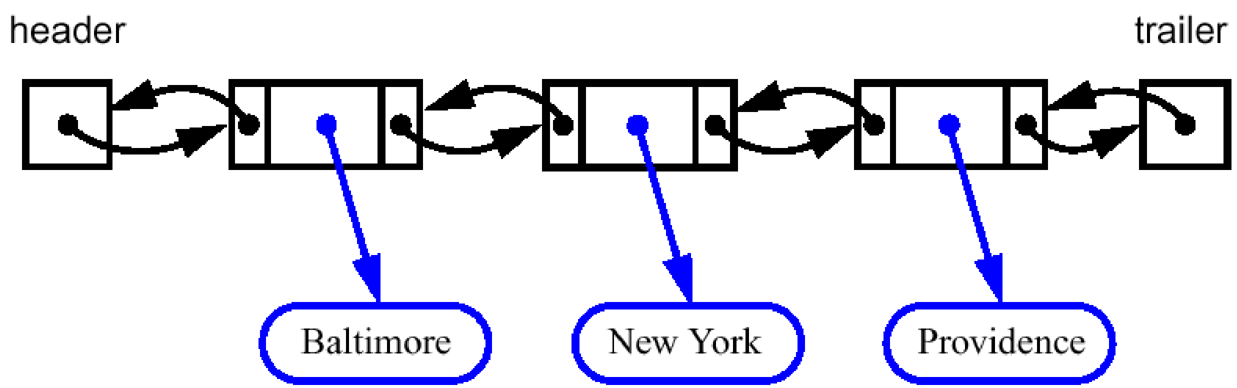
List with Explicit Head/Tail

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

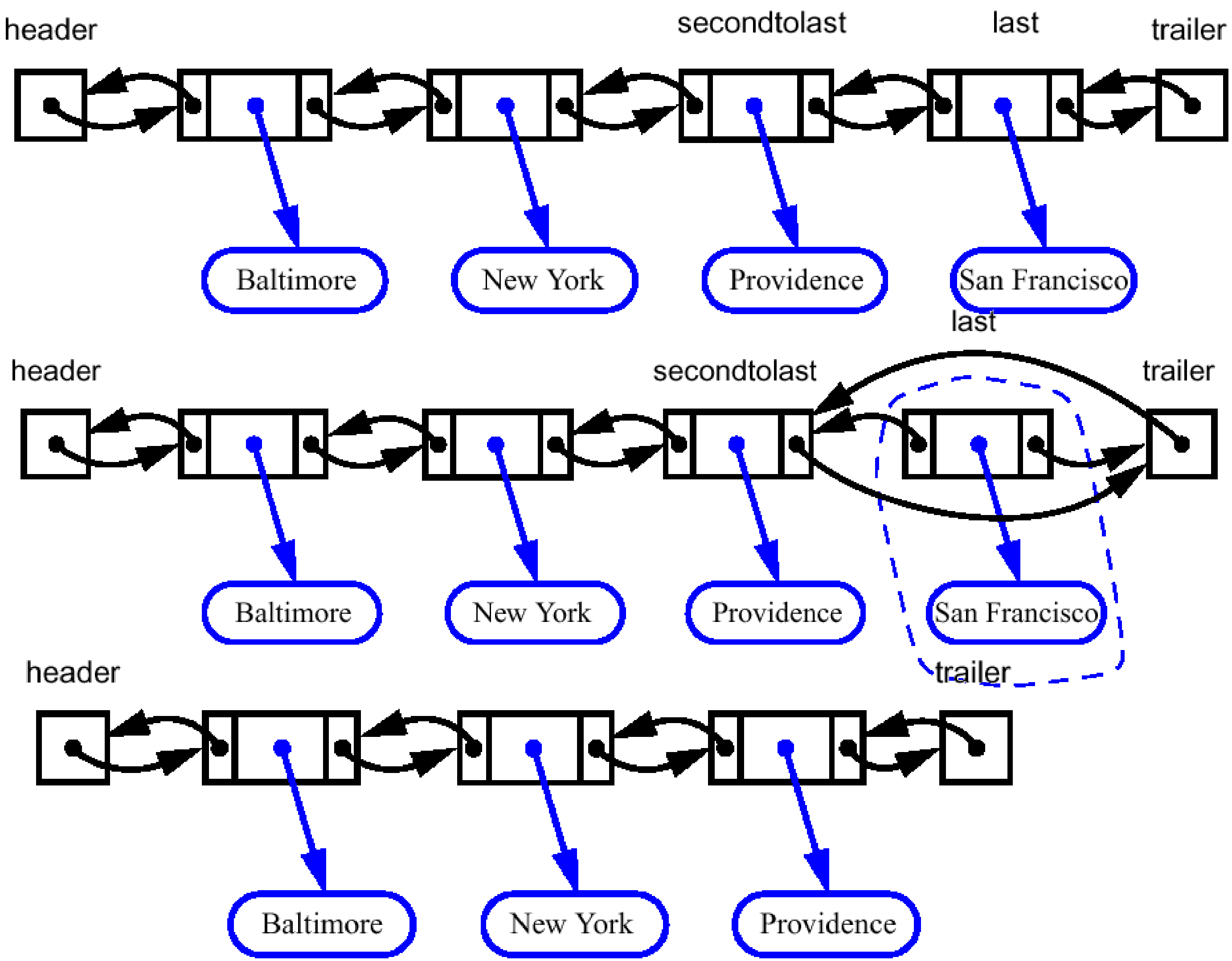


Doubly Linked Lists

- 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

DSA, Chapter 5: Overview

- 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:

- an *interface* – a collection of signatures of operations that can be invoked on an instance.
- a set of *conditions* (*preconditions* and *post-conditions*), possibly formulated as axioms,
  that define the semantics of the operations
  (i.e., what the operations do to instances of the ADT,
  but not how)

Examples of ADTs

We discuss a number of popular ADTs:

- Stacks
- Queues
- Priority Queues
- Ordered Lists
- Dictionaries (realized by Trees, next chapter)

They illustrate the use of lists and arrays

Why ADTs?

- ADTs allow one to break tasks 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.

Why ADTs?/2

- ADTs provide a language to talk on a higher level of abstraction

- ADTs allow one to separate *the check of correctness* and the *performance analysis:*
  1. Design the algorithm using an ADT
  2. Count how often different ADT operations are used
  3. Choose suitable implementations of ADT operations

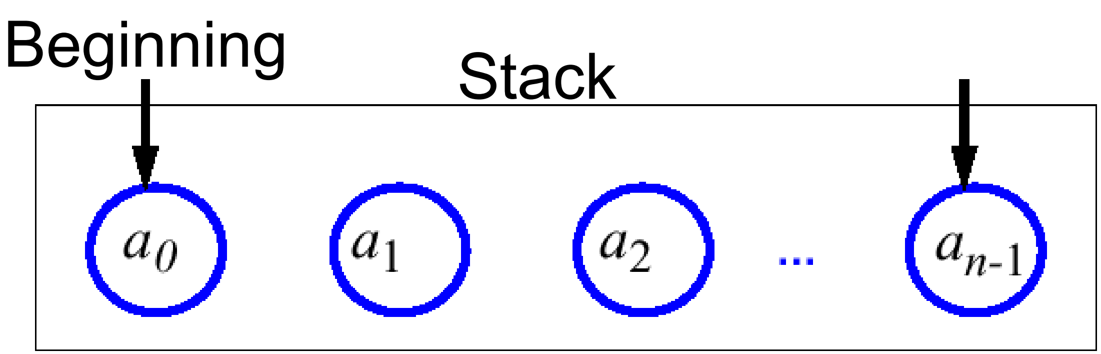
ADT = Instance variables + procedures
(Class = Instance variables + methods)

DSA, Chapter 5: Overview

- 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 is processed first
  - Example: OS stack, …
- Solution: Elements are inserted at the beginning (push) and removed from the beginning (pop)



Stacks/2

We assume

- there is a class *Element*
- we want to store objects of type Element in our stacks

We require that stacks support the operations:

- construction of a stack
  (possibly with a parameter for the maximal size)
- checking whether a stack is empty
- asking for the current size of the stack
- pushing an element onto the stack
- popping an element from the stack

Stacks/3

Appropriate data structure:

- Linked list, one head: 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 an 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

Array Implementation of Stacks

```
class Stack{
   int maxSize, count;
   Element[] S;

Stack(int maxSize){
   this.maxSize = maxSize;
    S = new Element[maxSize];
   count = 0; }

int size(){…}

boolean isEmpty(){…}

void push(Element x){ … }

Element pop(){ … }

}
```

Java-style implementation of stacks

Array Implementation of Stacks/2

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

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

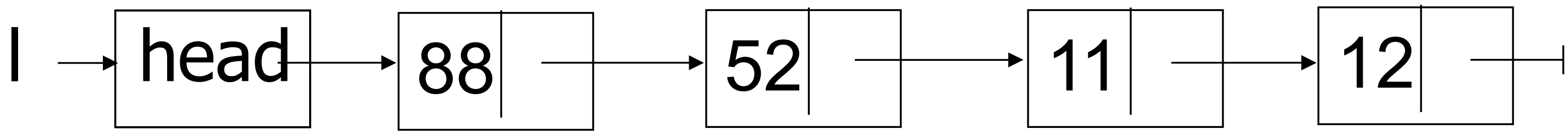
```
Element pop()
   if isEmpty() then Error
   x = S[count-1]
   count--;
   return x
```

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

Java-style implementation of stacks: arrays start at position 0

A Linked-list Implementation

- A list of integers:



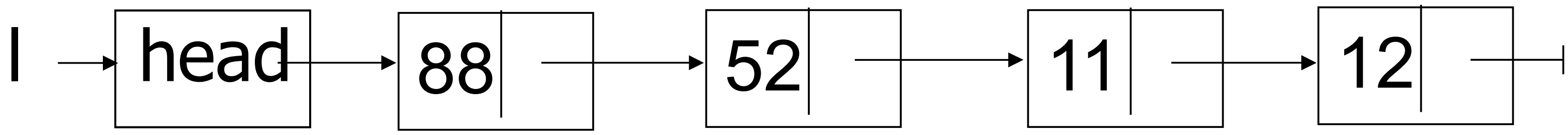
- Insert from the top of the list

```
void push(Element x):

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

- Constant-time operation!

A Linked-list Implementation/2

- A list of integers:



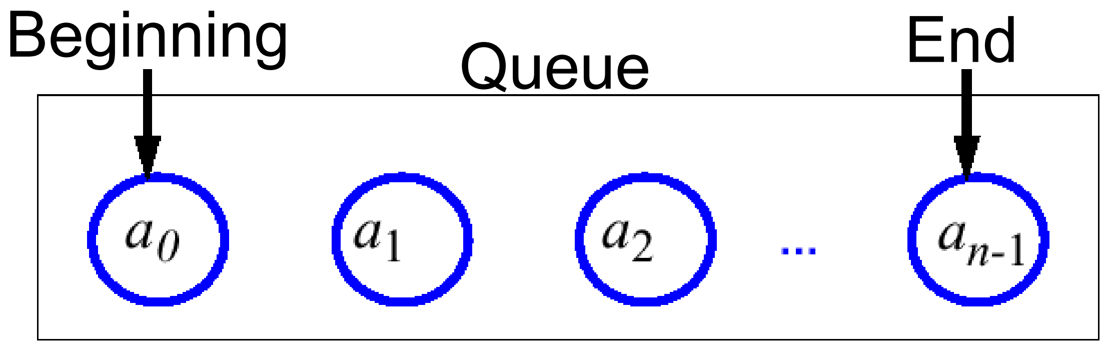
- Extract from the top of the list

```
Element pop():

x = head.val;
head = head.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 is processed first
  - Example: Printer queue, …
- Solution: Elements are inserted at the end (enqueue) and removed from the beginning (dequeue).



Queues/2

We assume

- there is a class *Element*
- we want to store objects of type Element in our queues

We require that queues support the operations:

- construction of a queue
  (possibly with a parameter for the maximal size)
- checking whether a queue is empty
- asking for the current size of the queue
- enqueuing an element into the queue
- dequeuing an element from the queue

Queues/3

Appropriate data structure:

- Linked list, head: 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 *maxSize* 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 (rear, for enqueuing)

An Array Implementation/2

"Wrapped around" configuration:

What does "*f* == *r*" mean?

An Array Implementation/3

In the array implementation of stacks

- we needed an array of size *N* to realize a stack of maximal size *N*
- we could model the empty stack with "**`count == 0`**"

Let's model a queue with an array of size *N* and "pointers" f, r:

- if f is fixed, then r can have *N* different values, one of them models "the queue is empty"
- hence, we can only store *N-1* elements, if we implement our queue with an array of length *N*

Array Implementation of Queues/3

```
class Queue{
   int N, f, r;
   Element[] Q;

Queue(int maxSize){
   this.N = maxSize + 1;
    Q = new Element[N];
   f = 0; r = 0;}

int size(){…}

boolean isEmpty(){…}

void enqueue(Element x){ … }

Element dequeue(){ … }

}
```

Java-style implementation of queues

An Array Implementation of Queues/4

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

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

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

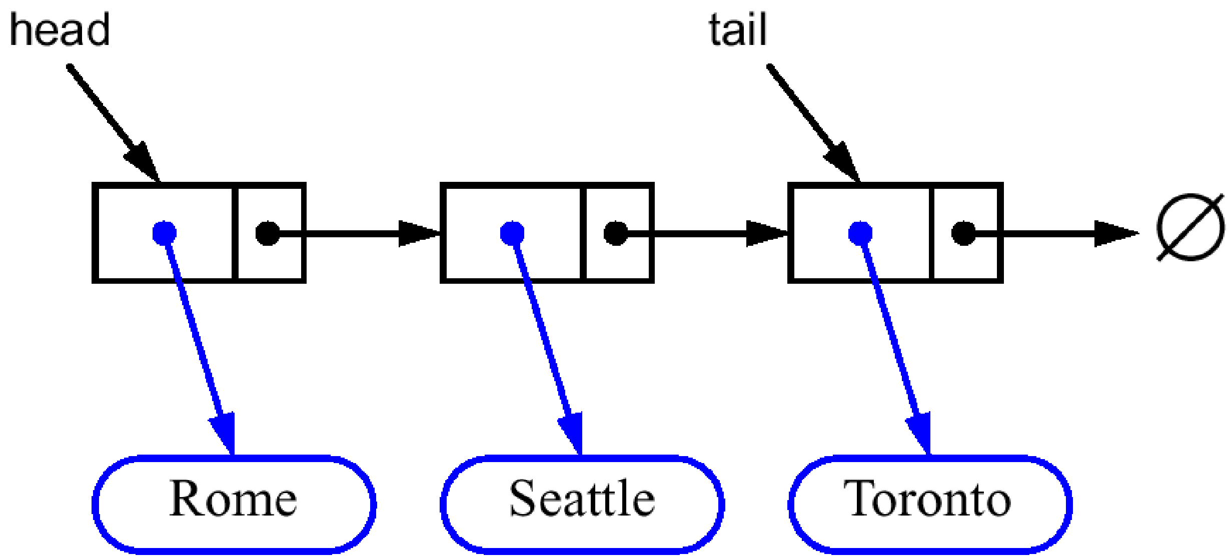
```
Element dequeue()
   if isEmpty() then Error
   x = Q[f]
   f = (f+1) mod N
   return x
```

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

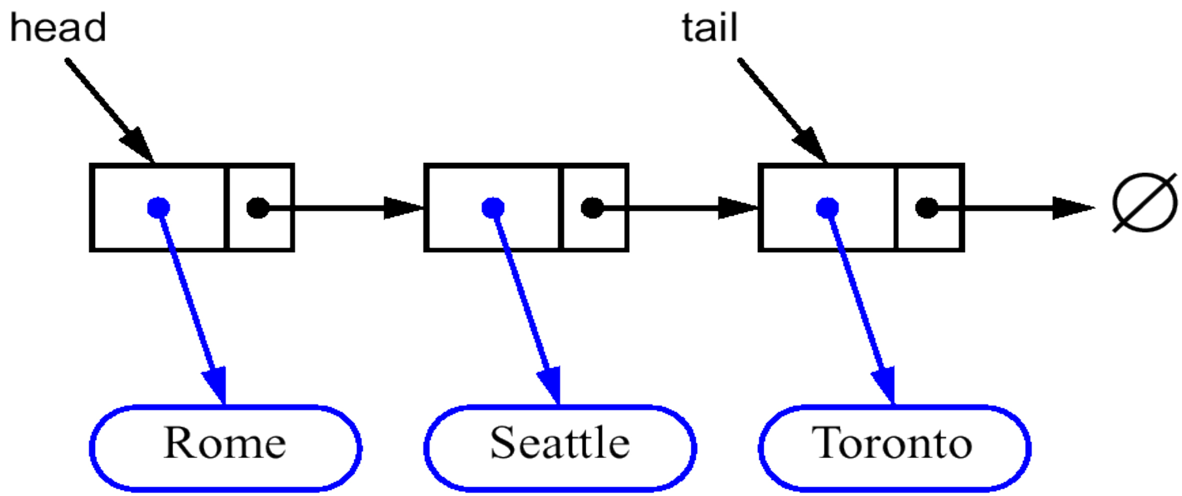
A Linked-list Implementation

Use linked-list with head and tail

Insert in tail, extract from head



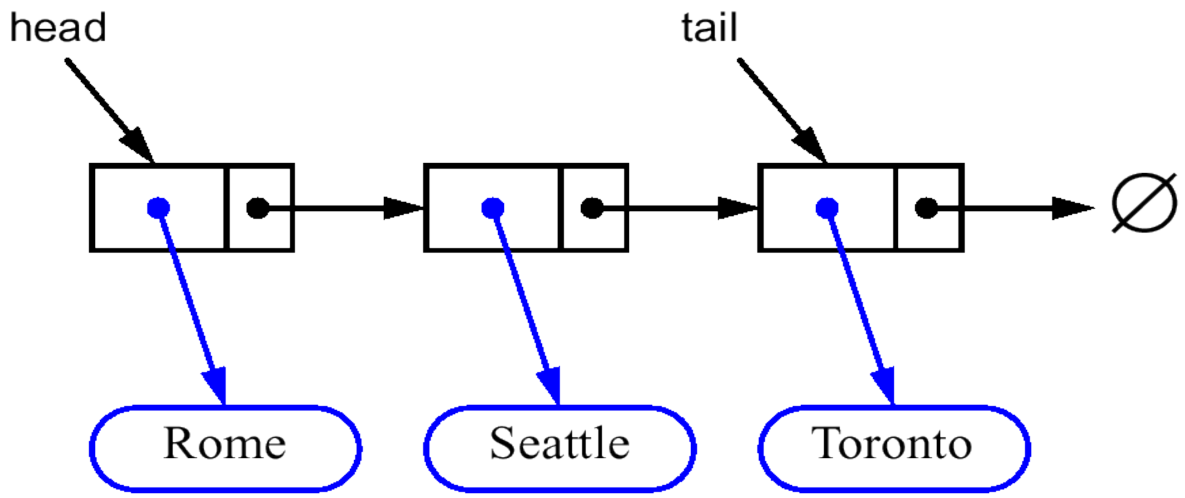
A Linked-list implementation/2



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

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

A Linked-list Implementation/3



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

```
Element dequeue():
x = head.info;
head = head.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)

DSA, Chapter 5: Overview

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

Ordered List

- In an ordered list Elements are ordered according to a key, which we assume to be an integer
- Example functions on ordered list:
  - `isEmpty()`
  - `int maxKey(), int minKey()`
  - `Element find(int key)`
  - `Element floorEntry(int key)`
  - `Element ceilingEntry(int key)`
  - `insert(int key, Element x)`
  - `print()`

Ordered List/2

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

```
class Node{
   int key; value Element;
   Node next;
}

class OList{
    Node head;
}
```

Ordered List/3

- Insertion into an ordered list (Java):

```
void insert(int i, Element x) {
  Node q = new Node();
    q.key = i; q.element = x; q.next = NULL;
  Node p;

  if (head == NULL || head.key > i) {
    q.next = head;
    head = q;
  } else {
...
```

Ordered List/4

Insertion into an ordered list (Java):

```
void insert(int i, Element x) {
  …
  } else {
    p = head;
    while (p.next != NULL && p.next.key < i)
      p = p.next;
    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

DSA, Chapter 5: Overview

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

Priority Queues

- A priority queue (PQ) 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

Array Implementation of Priority Queues

```
class PQueue{
   int maxSize, size;
   int[] A;

PQueue(int maxSize){
   this.maxsize = maxSize;
    A = new int[N];
   size = 0;}

int size(){…}

boolean isEmpty(){…}

void insert(int key){ … }

int extractMax(){ … }

}
```

Java-style implementation of priority queue of integers

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 PQs
  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 (head, head/tail, doubly linked)
- Abstract Data Types
  - Type + Functions
  - Stack, Queue
  - Ordered Lists
  - Priority Queues

Next Chapter

- Binary Search Trees
- Red-Black Trees