Software Structures and Models/Data Structures and Algorithms

TX00CK91

Lecture 5 - 14.04.2016

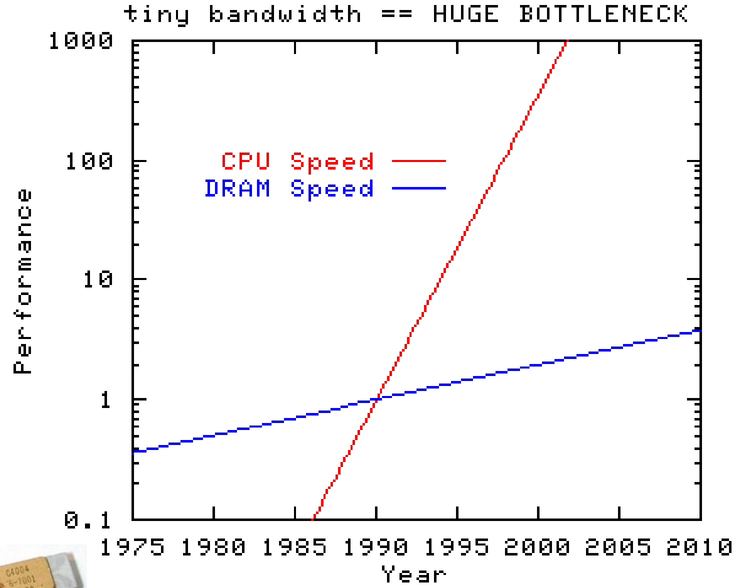
Jarkko.Vuori@metropolia.fi

Dynamic array

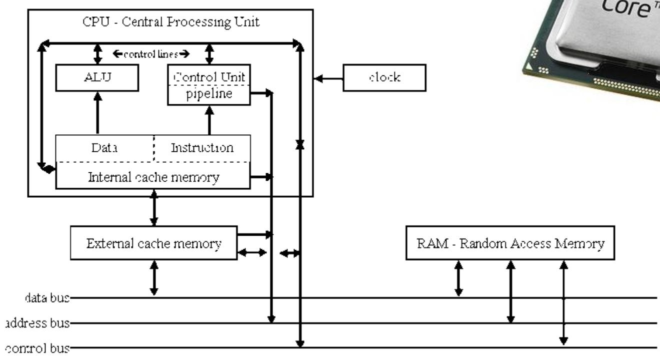
- The problem of the fixed size array could be solved by reserving a totally new array from the dynamic memory (by new) and copying all the elements from the old array to the new array (an array implemented in this way is a dynamic array)
  - The problem in this method is the copying of all the elements to the (larger) array
    - Copying is always a slow operation, therefore unnecessary data movement should always be avoided
  - The benefit of this method is that the size of the dynamic array can be changed during the time (application is running)
    - Compare this to the static array case where the size and type of the elements are fixed during the execution of the program

Why should we avoid moving of data

- Semiconductor memory technology has been mainly interested on increasing the size of the memory
  - Speed of the memory has not been the main goal
  - Compare this to the problem of the number of pixels vs. sensitivity and dynamic range in digital camera image sensors
    - There are more and more megapixels available on new cameras, but the sensitivity remains quite the same
- Processing power (of a microprocessor) has been doubled every year (according to the Moore's "law")
  - And the size of memory is doubling every 1.5 year
- For those reasons, the external bus (which connect the processor to the memory) is a performance bottleneck
  - Therefore the usage (e.g. copying the variables) of the external bus should be minimized
    - Partly this minimization is done automatically by cache memory



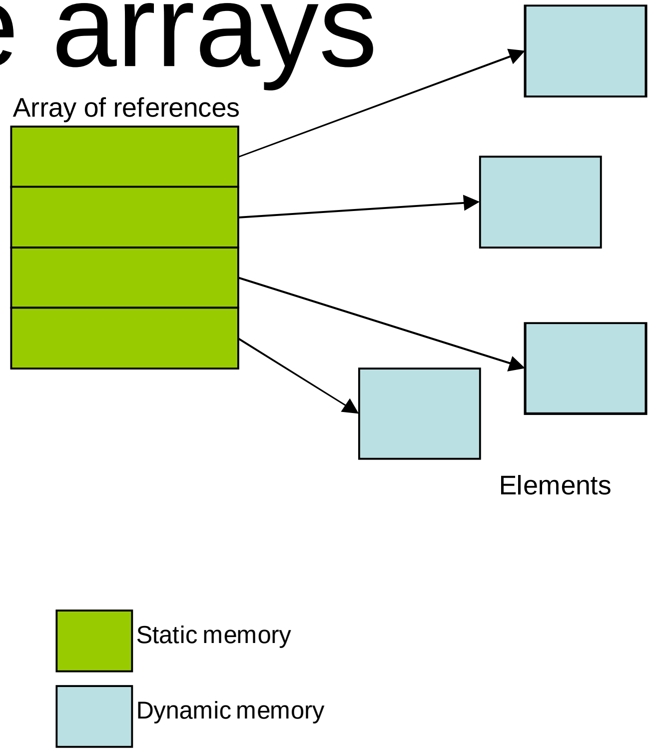
John D. McCalpin, University of Virginia



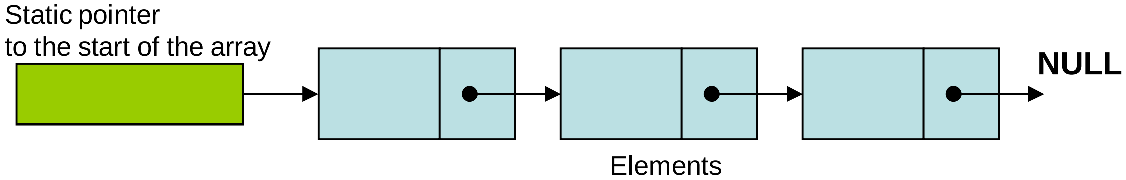
Arrays

- Static array
  - Fixed size limitation (either too much, or not enough)
  - When we run out of locations, nothing can be done
  - Inserting/deleting elements in the middle requires moving large part of the array
- Dynamic array
  - The size of the array can be changed during the run-time of the application
    - But this needs ”unecessary” data moving
  - When the size of the array is not enough, we can reserve a new larger array
    - And copy those old elements to the new array
- Drawbacks of the arrays in general
  - If the ordering in the array is maintained based on the content of the elements, insertions/deletions require movements in the array
- Benefits of the arrays
  - Random access based on the position number (index) is very rapid
  - Access based on the content is efficient, because we can use binary search ($\log_2 n$ operations needed to find an element from the group of n elements)
    - Elements are in order, then when we check the element in the middle, we can directly find out on what side the element we search for is
    - By splitting the right half, we go further to the element
      - And this can be done further again and again … to finally reach the needed element

Alternatives for the arrays

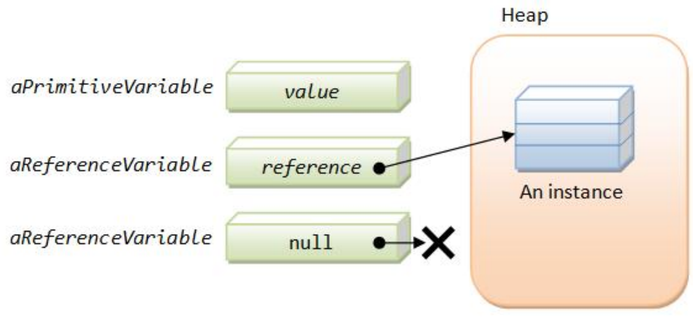


- Because there are drawbacks using arrays, we will develop an improved data structure; goals of our work are
  1. No upper limits for the number of elements
  2. No unnecessary space is reserved
     - neither too much, or not enough
  3. Unnecessary data movement never needed
     - Not needed even in insertion, deletion or size increasing
- Mainlines of the solution
  - Memory reservation
    - We reserve space for the element only when we need it
      - The problem is where to store all those references to those reserved memory locations
        - » We need those references because all the elements should be accessible, and objects are accessible only using the references
  - References
    - If the solution would be an array, we should know its size beforehand
    - When the array of references is full, we can't reserve more memory, because we don't have place where to store those references
- The solution is a linked data structure
  - Every element (reserved from the dynamic memory) contains the data and a link (reference) to the next element

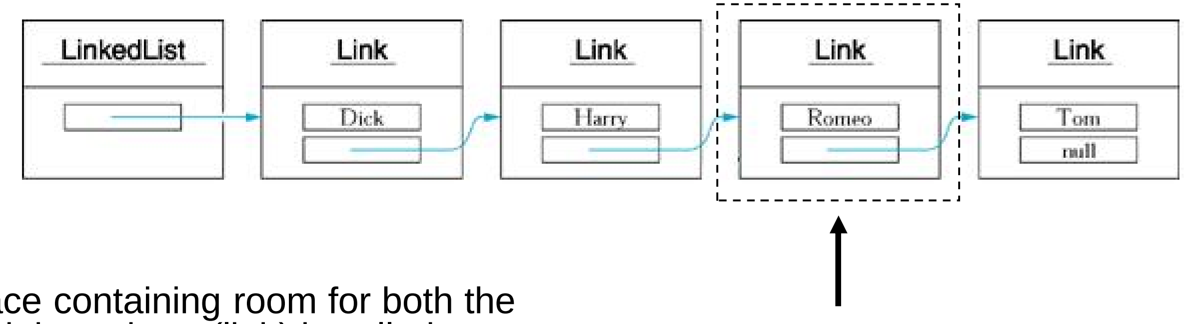


Java Reference Data Type

- Java follows the object reference model, which is followed by almost all OOP labguages
  - C++ differe from Java mainly because it does not follow the object reference model
  - Java does not use pointer, instead it uses references
- The reference variable are declared to refer to instances of a particular type and can only refer to instance of that type
  - E.g. `Rectangle r; Date d;`
    `d = new Date();`
  - The r and d above are not the object of type Rectangle and Date, they are simply references that can refer to an instance of Ractangle and Date
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it
  - This reference is, more or less, the address in memory of the object allocated by new



Dynamic, linked data structure



- The space containing room for both the data and the pointer (link) is called as a **node** (solmu)
  - Sometimes we call it as an element
- Node data type definitions and other definitions which are needed in linked data structure definition are shown on the right side

```
private class Node {
    private T data; // assuming generics here
    private Node next;

    public Node(T data, Node next) {
        this.data = data;
        this.next = next;
    }
}


// One node is created like this
last = new Node(item, null);
```

List as dynamic linked structure

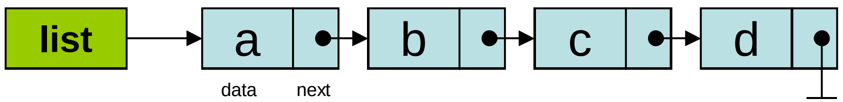
- Order between elements, this is expressed by links
  - In arrays, the position (of an element) expresses the ordering between elements
- Method 1 to implement a list
  One reference (first) represents the whole list
  - New typename LinkedList describes the ”new” role of the reference (as a list)
  - If we need the append operation (insert_to_list_end), then we need while-construction, where all the elements are traversed in order to find the last element (and then insert the reference of the new element to there)
    - It is not important, if we have order by the content type list
      - Because then in every cases we should traverse the list to find the place where to insert the new item
- Method 2 to implement a list
  - In this case the insert_to_list_end can be made efficient
    - Because the last node can be found directly with a reference

```
public class LinkedList<T extends Comparable<T>> {
    private Node first;

    public LinkedList() {
        first = null;
        last = null;
    }

     private class Node {
        private T data;
        private Node next;

        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
     …
}
```

```
public class LinkedList<T extends Comparable<T>> {
    private Node first;
    private Node last;

    public LinkedList() {
        first = null;
        last = null;
    }

     private class Node {
        private T data;
        private Node next;

        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
     …
}
```

List as dynamic linked structure, 1

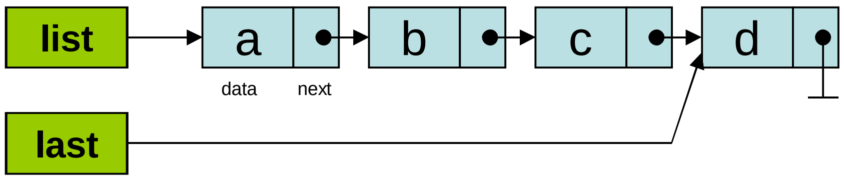
- Linear list, where the order is the sequence items are inserted to the list
- We implement a character list, in which the new character is always inserted to the end of the list
- We notice that the end of the list must be found repeatedly
  – Multiple use of references in succession
  – Using references is efficient, but multiple successive referencing operations is inefficient (takes time)
- Printing the list is easy and efficient, traversing the list one-by-one is simple with the next-field




```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node next;

    public Node() {
        this.next = null;
    }

    public static void main(String[] args) {
        Node<Character> list;

        list = new Node();
        list.data = new Character('a');

        list.next = new Node();
        list.next.data = new Character('b');

        list.next.next = new Node();
        list.next.next.data = new Character('c');

        list.next.next.next = new Node();
        list.next.next.next.data = new Character('d');

        Node what = list;
        System.out.print("List: ");
        while (what != null) {
            System.out.print(what.data + " ");
            what = what.next;
        }
        System.out.println();
    }
}
```

List as dynamic linked structure, 2

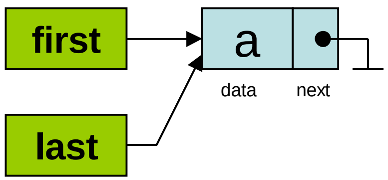
- Now we have an additional pointer for the last node of the list
- We still have multiple similar tasks when inserting a node




```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node next;

    public Node() {
        this.next = null;
    }

    public static void main(String[] args) {
        Node<Character> list, last;
        Node<Character> newnode;

        newnode = new Node();
        newnode.data = new Character('a');
        list = last = newnode;

        newnode = new Node();
        newnode.data = new Character('b');
        last.next = newnode;
        last = newnode;

        newnode = new Node();
        newnode.data = new Character('c');
        last.next = newnode;
        last = newnode;

        newnode = new Node();
        newnode.data = new Character('d');
        last.next = newnode;
        last = newnode;

        Node what = list;
        System.out.print("List: ");
        while (what != null) {
            System.out.print(what.data + " ");
            what = what.next;
        }
        System.out.println();
    }
}
```

List as dynamic linked structure

- Inserting node to the list end
  - First we create space for the node (in dynamic memory)
  - Copy the item data to the node
  - If the list is empty, set the node's reference to the first node (in static memory)
  - If there are already nodes on the list, take the reference of the last element and set it's next-field to refer to the newly created node
  - Set the last reference to refer to the new node, because it is now the last node of the list

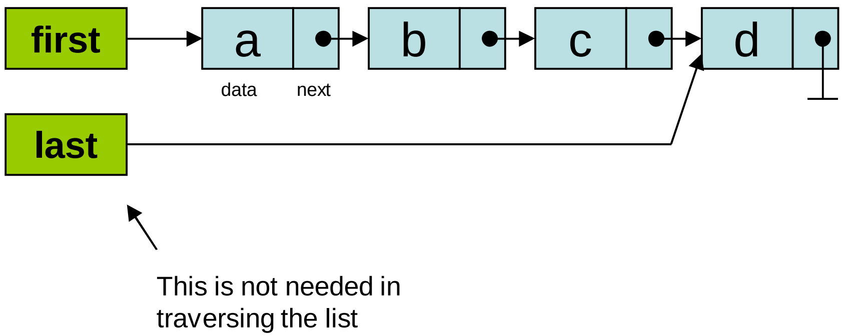



```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node next;

    private static Node first, last;

    public Node() {
        this.next = null;
    }

    public void insert_to_end(T item) {
        Node newnode = new Node();
        newnode.data = item;

        if (first == null)
            first = newnode;
        else
            last.next = newnode;

        last = newnode;
    }

    public static void main(String[] args) {
        Node<Character> newnode = new Node();

        first = null;
        newnode.insert_to_end(new Character('a'));
        newnode.insert_to_end(new Character('b'));
        newnode.insert_to_end(new Character('c'));
        newnode.insert_to_end(new Character('d'));

        Node what = first;
        System.out.print("List: ");
        while (what != null) {
            System.out.print(what.data + " ");
            what = what.next;
        }
        System.out.println();
    }
}
```

List as dynamic linked structure

- Traversing the linked list node by node is a quite efficient operation
  - As an example we show the printing of the elements in the list
- Because every node contains a reference to the next node, traversing all the elements is implemented by following the next-reference
  - End of the list is detected, when the next reference does not point to anything (NULL), i.e. there is no next node in the list

```
Node what = first;
System.out.print("List: ");
while (what != null) {
    System.out.print(what.data + " ");
    what = what.next;
}
System.out.println();
```



Class Linkedlist

- Separate variables first and last are grouped together (to a class Linkedlist)
  - Abstraction in class
- Operation functions for printing and inserting_to_end
- Constructor takes care of list initialization
  - cleaning (in order avoid memory leakage) is done automatically in Java by Garbage Collector

```
public class LinkedList<T extends Comparable<T>> {
    private Node first;
    private Node last;

    public LinkedList() {
        first = null;
        last = null;
    }

    private class Node {
        private T data;
        private Node next;

        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    …

    public static void main(String[] args) {
        LinkedList<Character> list = new LinkedList<Character>();

        list.add('a');
        list.add('b');
        list.add('c');
        list.add('d');

        System.out.println("List: " + list);
    }
}
```

Class Linkedlist

- Inserting item to the end of the list
  - Reserve space for the node
  - Copy item data to the node
  - If the first node of the list, assign its address to the first pointer
  - If not, assign its address to the last element's next-field
  - Update the last pointer
  - Terminate (set to NULL) the node's next –field
    - Last element does not have any successor elements

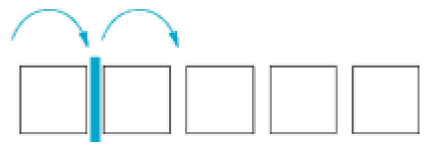
```java
public class LinkedList<T extends Comparable<T>> {
    private Node first;
    private Node last;

    public LinkedList() {
        first = null;
        last = null;
    }

    public void add(T item) {
        if (first != null) {
            Node prev = last;
            last = new Node(item, null);
            prev.next = last;
        }
        else {
            last = new Node(item, null);
            first = last;
        }
    }

    private class Node {
        private T data;
        private Node next;

        public Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
    }

    @Override public String toString() {
        StringBuilder s = new StringBuilder();
        Node p = first;
        while (p != null) {
            s.append(p.data + " ");
            p = p.next;
        }

        return s.toString();
    }
}
```

Abstract datatype (a recap)

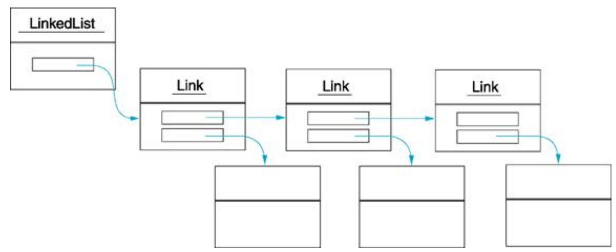
- ADT (Abstract Data Types)
  - Defines the basic data processing operations
  - Does not define the implementation
- ADT list
  - Ordering of the elements, elements can be traversed in this order
  - It is possible to insert/delete items to every place in the list
- ADT array
  - Ordering of the elements
  - Random access is possible when defining an integer index

Different views to a list and an array

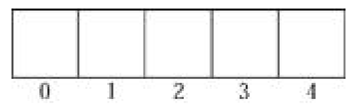
- Abstract view to the linked list



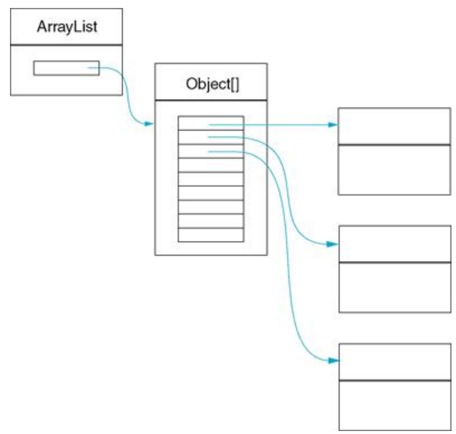
- Concrete view to the linked list



- Abstract view to the array



- Concrete view to the array



Properties of list and array

- Complexity of the linked list
  - Interting/deleting an item
    - Constant number of pointer references must be modified in order to insert/delete an item, independent of the size of the list
    - Item can be inserted/deleted in constant time
    - In big-Oh notations: O(1)
  - Random access
    - On the average, half ($n/2$) of the elements in the list must be traversed
    - In big-Oh notation: $O(n)$
- Complexity of the array list
  - Inserting/deleting an item
    - On the average, half (n/2) of the elements in the array must be moved
    - In big-Oh notations: $O(n)$
  - Random access
    - In big-Oh notation: O(1)

Big-O notation, O(n), is a mathematical notation used to describe the asymptotic behavior of functions. If for example, memory consumption is $T(n)=4n^2-2n+3$, and if the term n is large, term n^2 dominates. Then the memory consumption is $O(n^2)$

Complexities of list and array

Operation	Array	List
Random access	O(1)	O(*n*)
Accessing the next item	O(1)	O(1)
Deleting/inserting an item	O(*n*)	O(1)