Software Structures and Models/Data Structures and Algorithms

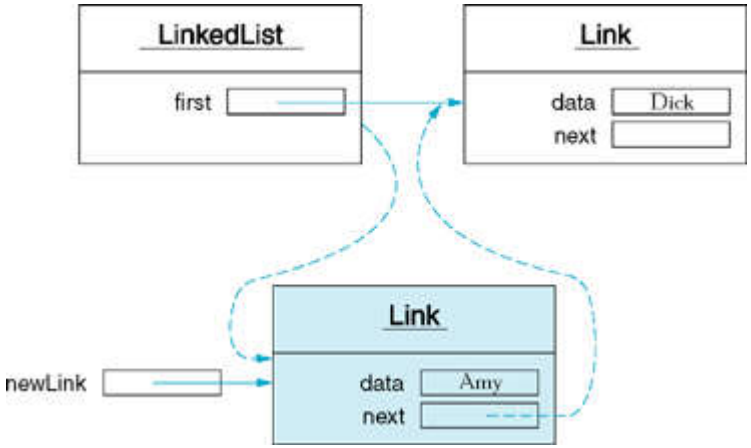
TX00CK91

Lecture 06 - 19.04.2016

Jarkko.Vuori@metropolia.fi

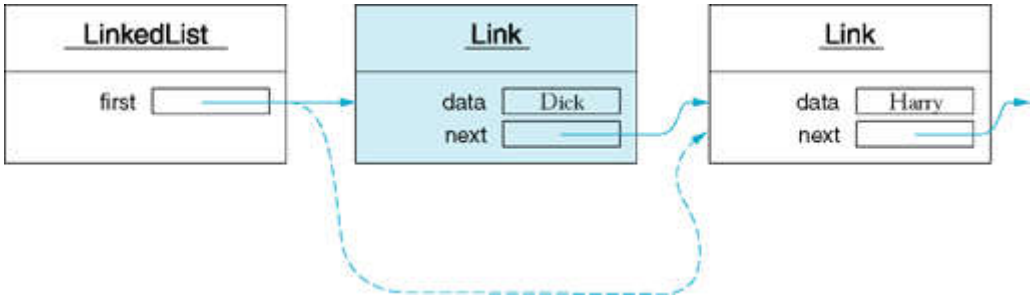
Inserting a node to the list start

- Assume that the list is defined as `private Node first;`
- Inserting a node to the beginning of the list is very efficient, because no repetition constructions (for, while, …) are needed
  - No ”unnecessary” copying/moving of data
- This kind of data structure can be used as a data storage
  - In this case we are not interested in the ordering of the nodes
  - Or when data is needed in reversed order
    - New data is inserted to the beginning of the list
    - When reading (from the list) we get the first entered data last




```
public void insert_to_begin(T item) {
    if (first != null) {
        first = new Node(item, first);
    }
    else {
        first = new Node(item, null);
    }
}
```

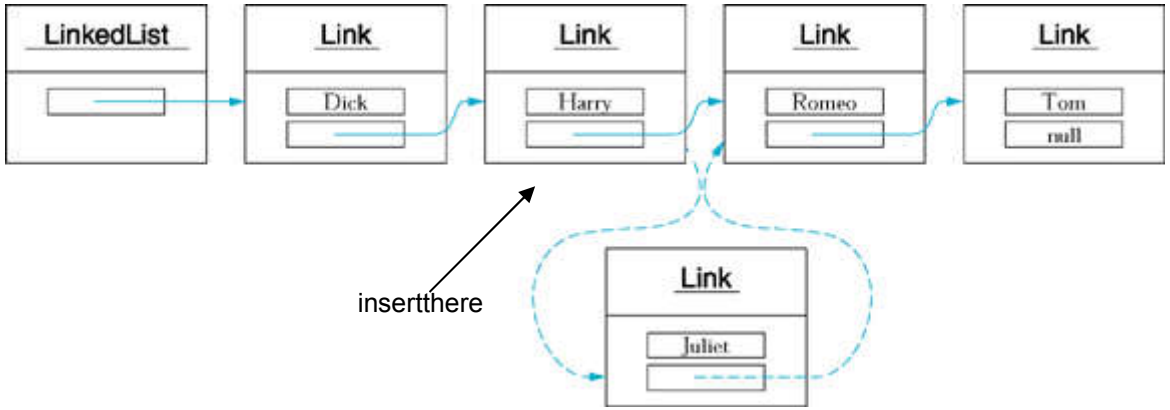
Removing a node from the beginning of the list



- Value of the next-field of the first node on the list is assigned to the list's first pointer
- The old first node is released later automatically by Java garbage collector

```
void delete_from_list_head() {
    first = first.next;
}
```

Inserting a node to the middle of the list



- First create a new node, assign insertion point's next pointer value to the new node's next pointer
- Then, set the insertion point next link to point to the created node

```
void insert_to_list_middle(Node inserthere, T item) {
     Node newnode;

     newnode = new Node(item, inserthere.next);

     inserthere.next = newnode;
}
```

Dynamic, linked stack implementation

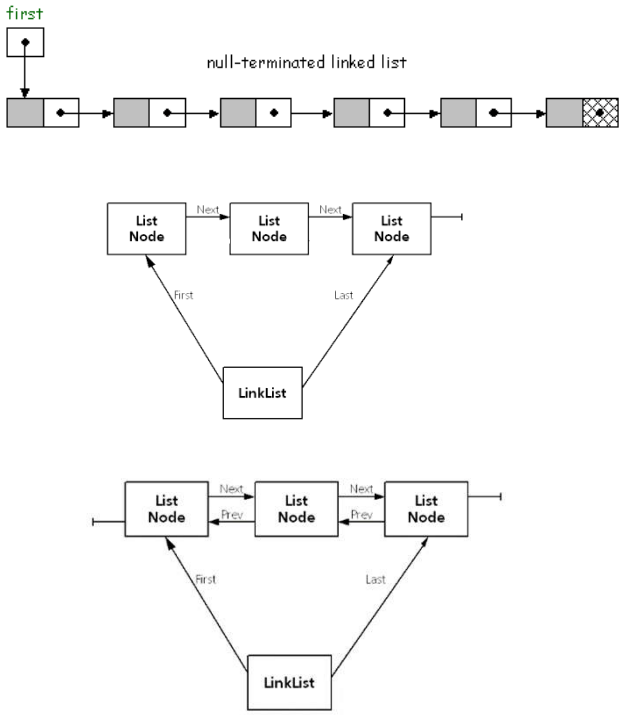
- We want stack container to be implemented using a linked list
  - Then we don't have the size limitation and there is no data movement (in case of making the array larger)
- Inserting to the beginning of the list is an efficient operation
  - Accessing this lastly inserted node is also an easy operation
    - Because it is now the first element of the list
- Deleting this lastly inserted node is also an easy operation
  - Because we have the pointer to this node
- We need two operations in the stack: push and pop
  - They can be implemented with the list insertion (to the beginning of the list) and the list deletion (from the beginning of the list) operation functions
  - The efficient way to implement the stack was found by an accident
    - Or by an "educated guess"

Problems with this accidental development method

- You can find the right answers by an accident (with instinct, emotions, intuition, etc.), but in most cases those answers are (at least partly) wrong
- If I want my coffee to cool rapidly, should I mix a cream immediately to it, or after a while?
- Can I avoid to be wet, if I run rapidly to the shelter?

Dynamic, linked stack implementation

- Structure of the stack can be also found with careful engineering thinking
  - Attempt 1: Stack is a special case of linked list, links represent the order of insertion, new data is inserted to the end
    - Push and pop –operations require the traversal of every element
  - Attempt 2:
    - We use the linked list version 2 where also the address of the last element is stored
    - Push-operation is simpler, but pop needs the traversal of the elements
      - Because when deleting from the list end, the node before the last element must be found
        - » In order to update the last pointer
  - Attempt 3:
    - We use two directional linking (node has an additional link to the previous node)
    - Additional (backward) links consumes memory, and complicates the logic of operation
      - How to update the links when a node is removed?
  - Attempt 4:
    - We continue with the attempt 1, but reversing the link direction
    - Now Pop and push are simple an efficient
  - Engineering methodology is disciplined (but not necessarily tedious), and the right solution (simple and efficient) was found



All the different possibilities are tried and analyzed, then the best solution is selected

Dynamic, linked stack implementation

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

    public LinkedStack() {
        first = null;
    }

    public void push(T item) { // insert_to_begin
        first = new Node(item, first);
    }

    public T pop() { // delete_first
        Node temp = first;

        if (first != null)
            first = first.next;

        return temp.data;
    }

    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();
    }
```

```
 public static void main(String[] args) {
        LinkedStack<Character> stack = new LinkedStack();
        Character             item;

        System.out.println("Enter a letter to push onto stack");
        System.out.println("or digit 1 to take a letter from stack");
        System.out.println("Return to end the program\n");
        try {
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('\n')) != 0) {
                if (item.compareTo(new Character('1')) == 0)
                    System.out.println("Letter popped from stack is " + stack.pop());
                else
                    stack.push(item);

                System.out.println("Stack content: " + stack);
                item = new Character((char)System.in.read());
            }
            System.out.println();
        } catch(Exception e) {
            System.out.println("Exception " + e);
        }
    }
}
```

Linked list queue implementation

- How about a queue with the linked list concept
  - Attempt 1: implementation 1
    - Not good, reversal of the link does not help
      - Because we can fetch efficiently only the pointer we last inserted

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

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

    …
```

  - Attempt 2: implementation 2
    - Good solution, insertion is efficient
      - Insert to the list end
      - Uses `LinkedQueue` structure's last field
    - Removal from the queue is also efficient
      - We remove from the beginning of the list
      - Works as take_beginning but with using the first field

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

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

    …
```

Linked list queue implementation

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

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

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

    public T dequeue() { // delete_first
        Node tmp = first;

        if (first != null) {
            first = first.next;
            if (first == null)
                last = null;
        }

        return tmp.data;
    }
```

```
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();
}
```

Linked list queue implementation

```
 public static void main(String[] args) {
        LinkedQueue<Character> queue = new LinkedQueue<Character>();
        Character              item;

        System.out.println("Enter a letter to enqueue onto queue");
        System.out.println("or digit 1 to dequeue a letter");
        System.out.println("Return to end the program\n");
        try {
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('\n')) != 0) {
                if (item.compareTo(new Character('1')) == 0)
                    System.out.println("A letter dequeued " + queue.dequeue());
                else
                    queue.enqueue(item);

                System.out.println("Queue content: [" + queue);
                item = new Character((char)System.in.read());
            }
            System.out.println();
        } catch(Exception e) {
            System.out.println("Exception " + e);
        }

    }
}
```

Point of generality

- Linked list implementations of the stack and queue are compatible (in a higher abstraction level) with previous array implementations
  - If we need dynamic stack and queue containers, we can change our applications to use the linked list implementation without any modifications to the source code
- Linked list container implementations are general purpose solutions considering the item to be stored
  - Because the implementation is created using templates

Linked list and array

- Linked list is an a alternative for a array
- Array and a linked list can be combined
  - Data element in linked list can also be an array
    - Static: `LinkedList<int [100]> list;`
    - Dynamic: `LinkedList<int *> list;`
  - Linked list can be as an element in an array
    `LinkedList<int> array[100];`

    Other linked list can also be an item in linked list
    - sparse matrix

Array

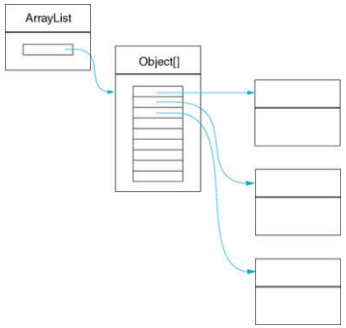
addr	value
C0	"Carol"
C1	"Bob"
C2	"Alice"
C3	0
C4	0
C5	0
C6	0
C7	0
C8	0
C9	0
CA	0
CB	0

main memory

Linked list

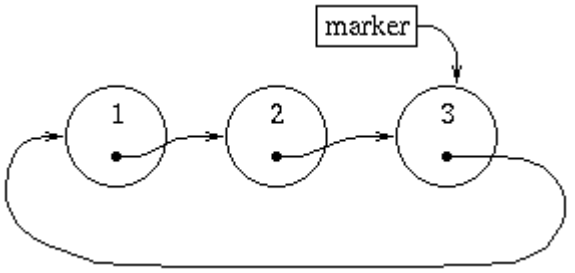
addr	value
C0	"Alice"
C1	null
C2	0
C3	"Carol"
C4	C9
C5	0
C6	0
C7	0
C8	0
C9	"Bob"
CA	C0
CB	0

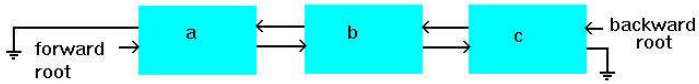
main memory

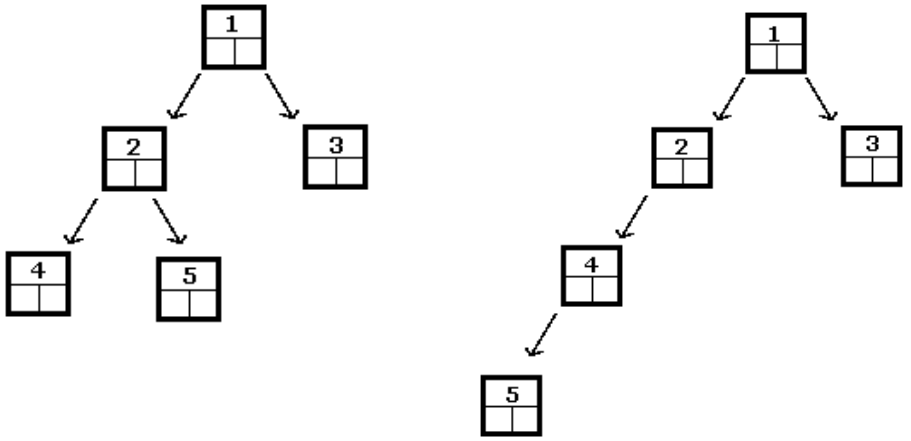


Other dynamic datastructures

1. Linked ring
2. Doubly-linked list
3. Tree structures
   - Binary search tree
   - Balanced tree





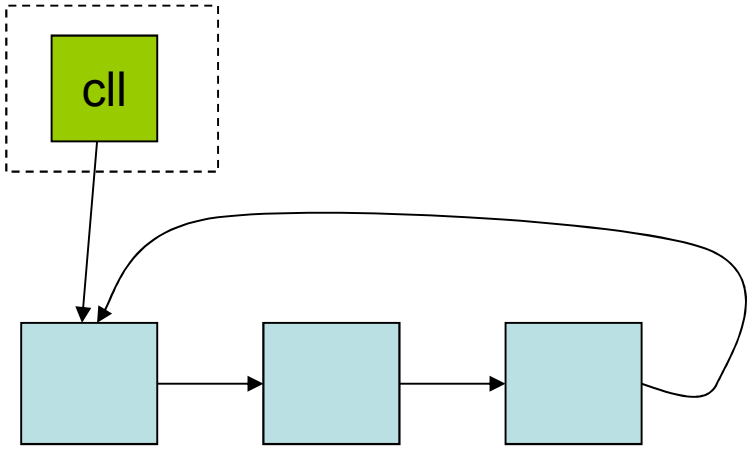


Linked ring

- Functionally equal to the ring buffer (or ring queue)
  - But the size of the ring is dynamic
- Inserting a new item to the ring is efficient
- Applications
  - Previous N actions storage, e.g. maintaining a log
  - Running average of measurements (N previous measurements are easily available, because we can easily get from the last node to the first node of the ring)
- Linked ring as an ADT
  - Implementation 1: ring represented by one pointer
    - In linked ring, the last node will point to the first node in the ring
    - Last node is not separately indicated
      - We don't find that last node with a NULL value in the field next
      - Instead last node is indicated when the next field (in the last node) has the same address than the first node
  - Internal representation may also have the number of nodes
    - Then counting the number of nodes is easy
  - New measurement (not a new node) is inserted to the list as follows:

```
cll.data = new_value;
cll = cll.next;
```

    - Note that cll refers always to the last node in the ring
- Possible operator functions:
  - Initialize (the ring)
  - The number of items (nodes) in the ring
  - Insert item to the ring
  - Delete item from the ring
  - Traverse all items in the ring



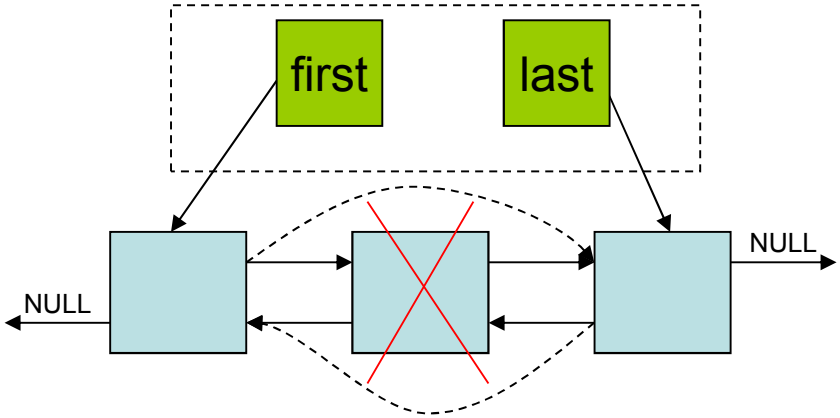

```
private class Node {
    private T data;
    private Node next;

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

Doubly linked list

- Doubly linked list has links to both directions (forwards and backwards)
- Applies to whatever cases, where traversing to both direction is needed
  - E.g., when we want to make searches to both directions (relative to the current node)
    - Find previous, find next
  - e.g., when we are handling unlimited lenght numbers
    - We need traversing from left to right (input and output)
    - And also traversing from right to left is needed (addition)
- Many operations are simpler (deleting a node from the end of the list)
  - Node insertion is a little more complicated

Doubly linked list

- Delete a node whose address is in a pointer p
  - Previous and next nodes are easily found
- What about the cases
  - Empty list
  - Only one element in the list
  - Delete the first node
  - Delete the last node




```
class Node {
    T    data;
    Node next,
         prev;
}

void delNode(Node p) {
    p.prev.next = p.next;
    p.next.prev = p.prev;

   delete p;
}
```

Sparse matrix

- Spread sheet programs (e.g. MS Excel) gives user a matrix which has huge amount of cells (even million cells)
  - But only a small part of that is really used in most cases
  - It is not efficient to reserve all that memory at once; memory reservation should be done only when the cell is really used (dynamic memory management)
- This same principle can be used to represent sparse matrix in general

  5 0 0 1
  0 3 0 0
  0 0 6 7
  0 0 0 9
- We can save memory if zero elements are not stored in memory

```
void read ();
void print ();
int  if_square();
float determinat();
Matrix add(Matrix a, Matrix b);
Matrix multiply(Matrix a, Matrix b);
```

```
// definitions for linked lists
// containint rows
// (one row is a list of matrix cells)
class Row {
  T    item;
  int  colnumber;
  Row  next;
};

// definitions for linked list
// containing matrix cells in one row
class Col {
  Row   row;
  int   rownumber;
  Col   next;
};
```