CP2 Revision

theme: linked lists

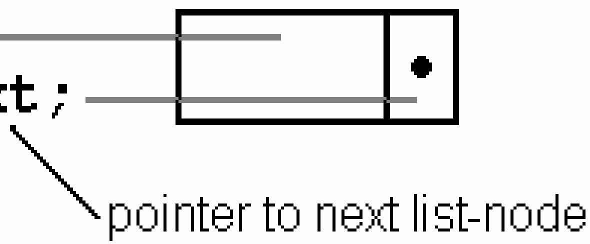
linked lists

- dynamically created during program run-time
- sequential collection of self-referential elements (*called* ***nodes***)
- elements are accessed linearly by sequentially traversing the list from start to finish

linked lists (2)

- number of nodes can grow or shrink dynamically
- each node resides in a separate place in memory (*not continuous*)
- Example: **singly linked list node**

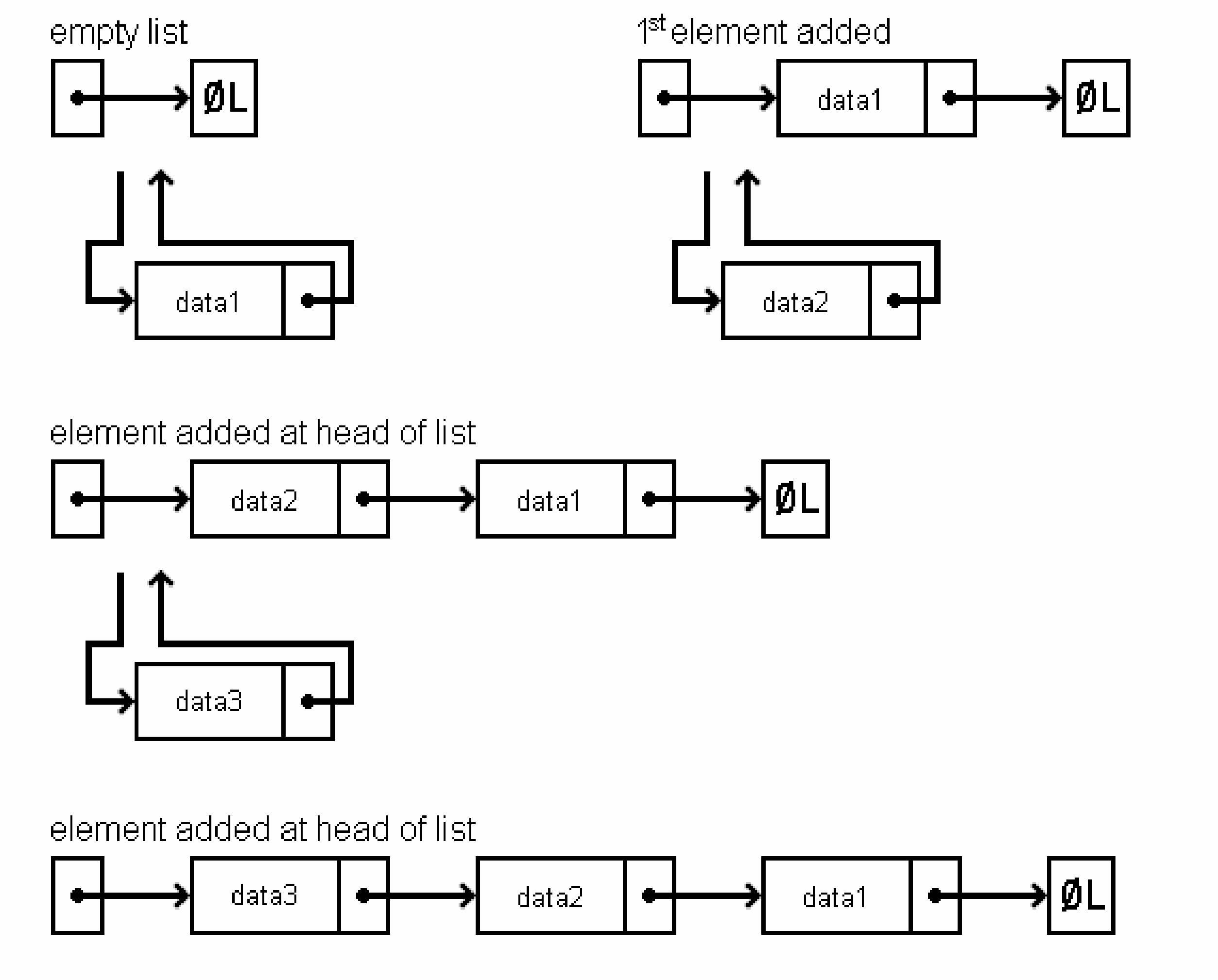
```
typedef struct _node
{
  int data;
  struct _node* next;
} node;
```



linked lists (3)

- each list element (*node*) not only has a data member but also a pointer connection (***link***) to the next list element
- lists are linear collections of data, i.e a list node always has only one predecessor and only one successor
- by convention the final link in a list (*link in last node*) is set to NULL
- list is accessed through a pointer to the first node of the list (*base pointer*)

(*singly*) linked list construction



singly linked lists (3)

- by convention the functions for handling a linked list are called
  - **insert** for adding a list element
  - **delete** for removing a list element
- sometimes two more functions are used
  - **isEmpty** to check if list is empty
  - **printList** to print the whole list

linked list access

- by convention the functions for handling a linked list are called
  - **insert** for adding a list element
  - **delete** for removing a list element
- sometimes two more functions are used
  - **isEmpty** to check if list is empty
  - **printList** to print the whole list

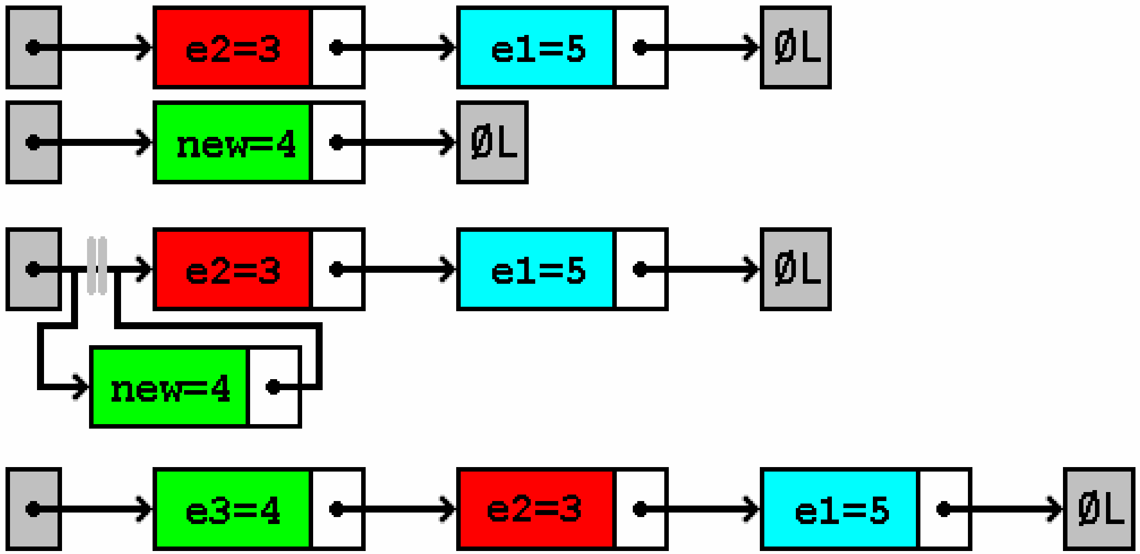
linked list access (*2*)

- all functions can be implemented in different ways
- one can use iterative or recursive methods for list traversal
- list elements can be inserted at different positions in the list
  - head
  - tail
  - centre (*in an ordered list*)

```
typedef node* nodePtr;

int insert(nodePtr*,int);
```

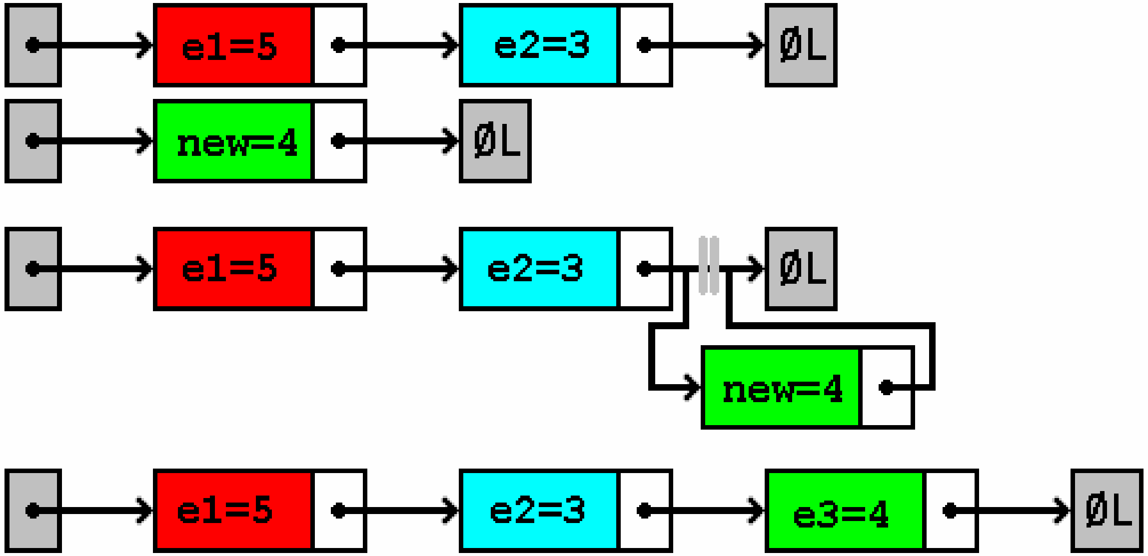
inserting at head of list



```
typedef node* nodePtr;

int insert(nodePtr*,int);
```

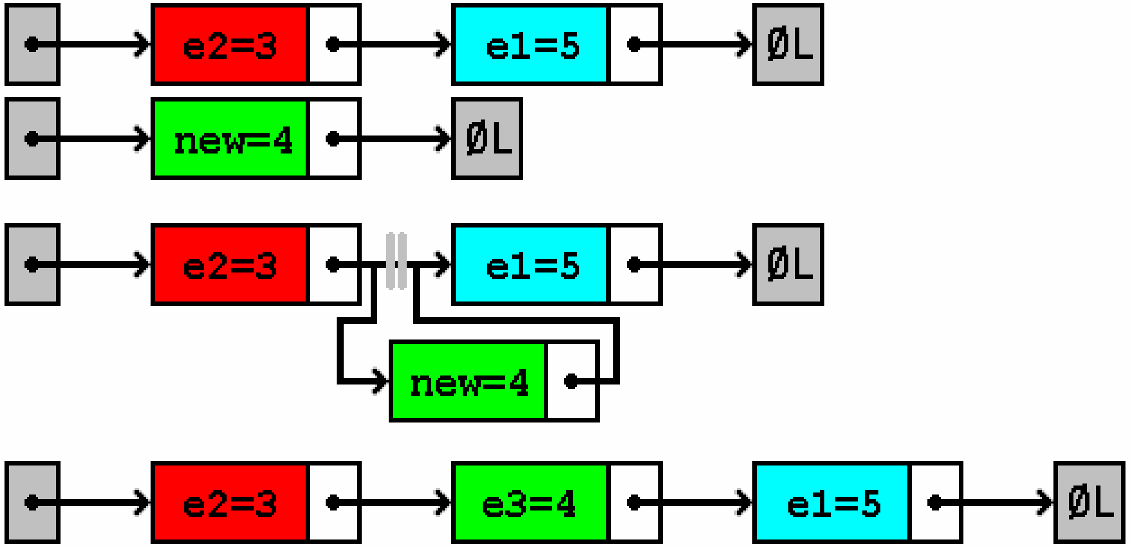
inserting at tail of list



```
typedef node* nodePtr;

int insert(nodePtr*,int);
```

inserting into an ordered list



deleting list elements

- problems:
  - remaining list must stay intact
  - node memory must be freed (*deallocated*)
- solution can be iterative or recursive

```
/* delete a list element (recursive solution) */
int delete(nodePtr *current, int value)
{
  nodePtr temp;
  if(isEmpty(*current))       /* catch error, before it occurrs */
    return 0;
  if(value==(*current)->data) /* if the element is current node */
  {
    temp=*current;        /* get the address of current element */
    *current=(*current)->next;     /* update the list structure */
    free(temp); /* free memory used by the data that is deleted */
    return value;              /* 'report' back to the program */
  }
  else                /* keep searching the list for the element */
    return delete(&((*current)->next),value);
}
```

problems with singly linked lists

- only one-directional (*linear*) sequential access
- worst case (*of necessary*) steps for finding element is no. of list nodes in list ***n***

doubly linked lists

- datastructure for bi-directional sequential access
- functions are same as for singly linked lists (*with modifications*)
- each node has 2 links:
  - one to the next node (*next link*)
  - one to the previous node (*previous link*)
- allows traversing of linked list forwards and backwards

doubly linked lists (*2*)

- next link of last node points to **NULL**
- previous link of first node points to **NULL**

Usually implemented like this:

```
struct _node
{
  struct _node *next; /* link to next */
  struct _node *prev; /* link to previous */
  int data;           /* node data */
};
typedef  struct _node node;
typedef  node   *nodePtr;
```

doubly linked lists (*3*)

- Inserting and deleting becomes more complicated:
  - 2 links must be reconnected correctly
  - this means: addidional special cases

linked list based datastructures

- stacks
- queues

emulating stacks with linked lists

- **FILO** (***f****irst-****i****n* ***l****ast-****o****ut*) / **LIFO** (***l****ast-****i****n* ***f****irst-****o****ut*) datastructure
- data can only be added and/or removed from top of stack (*head of the list*)

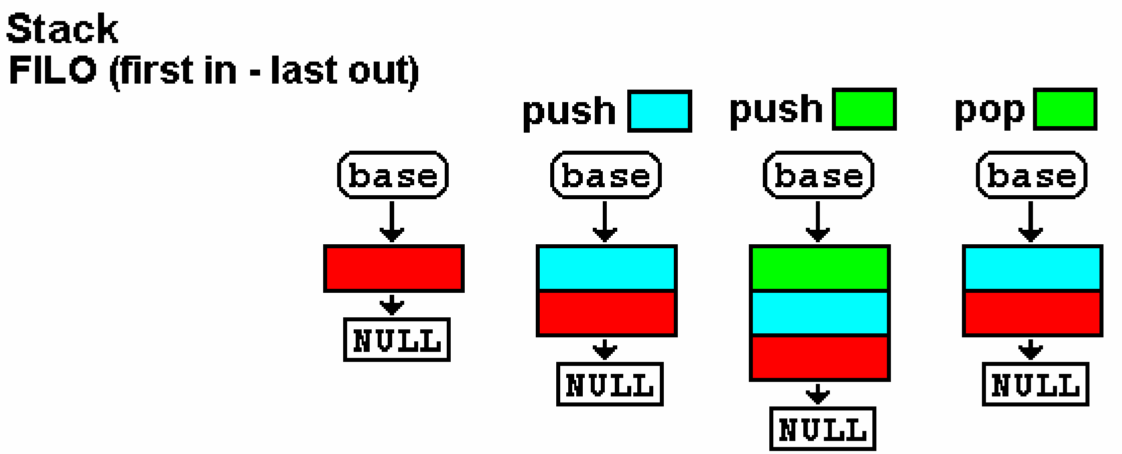
Functions used with stacks:

- **push** adds an element to top of the stack (*base of the list*)
- **pop** removes an element from top of the stack

sometimes there is also a function

- **top** a pop immediately followed by a push

stack




```
int push(nodePtr* head,int data)
{
  nodePtr newNode;
  newNode=(nodePtr)malloc(sizeof(node));
  if(newNode!=NULL)
  {
    newNode->data=value;
    newNode->next=*head;
    *head=newNode;
    return 1;
  }
  else return 0;
}
```

```
int pop(nodePtr* head)
{
  nodePtr temp;
  int retval;
  temp=*head;
  if(temp==NULL)
    return 0;
  retval=temp->data;
  *head=temp->next;
  free(temp);
  return retval;
}
```

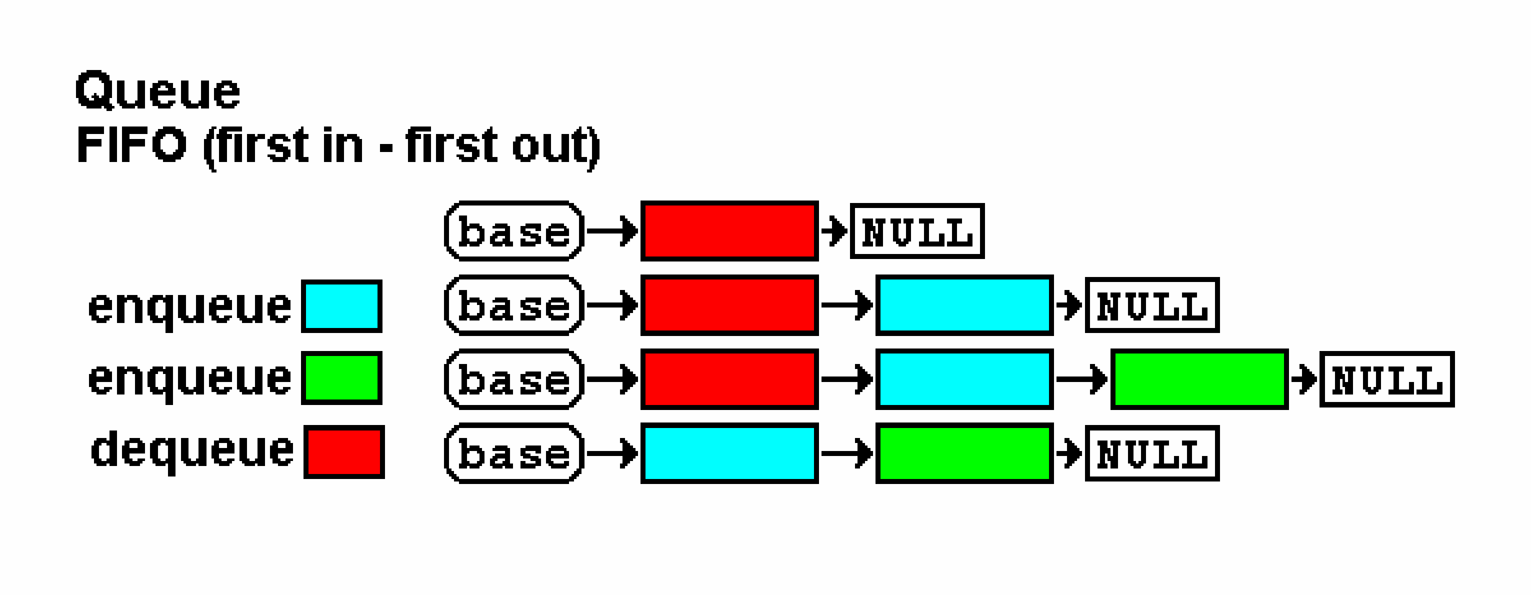
emulating queues with linked lists

- **FIFO** (***f****irst-****i****n* ***f****irst-****o****ut*) datastructure
- data can only be entered at the end of the queue (*tail of the list*)
- data can only be removed from the start of the queue (*head of the list*)

Functions used with queues:

- **enqueue** adds an element to end of queue
- **dequeue** removes an element from start of queue (*base of the list*)

queues



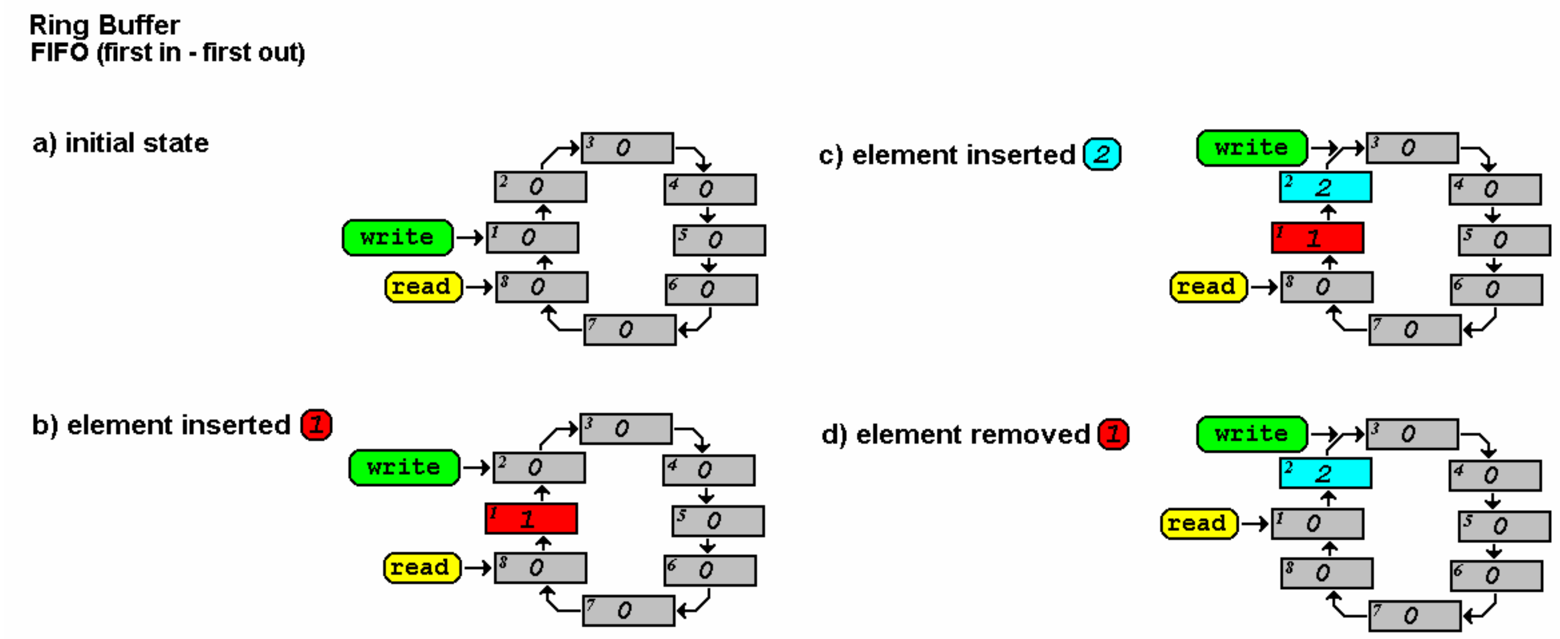
- **enqueue** - identical to list insert at tail
- **dequeue** - identical to stack's pop

a different kind of queue

circular buffer / ring buffer

- variation of the queue (*list*)
- no first or last element - tail points to head
- has static number of elements - does not grow or shrink
  - nodes are generated (allocated) at program start
  - Nodes are freed at program end

circular buffers



- ***2*** base pointers (*write-pointer & read-pointer*)
- **writing**: enter data & advance write pointer
- **reading**: advance read pointer & retrieve data