Elementary Structures & Memory Management

Juan Mendivelso

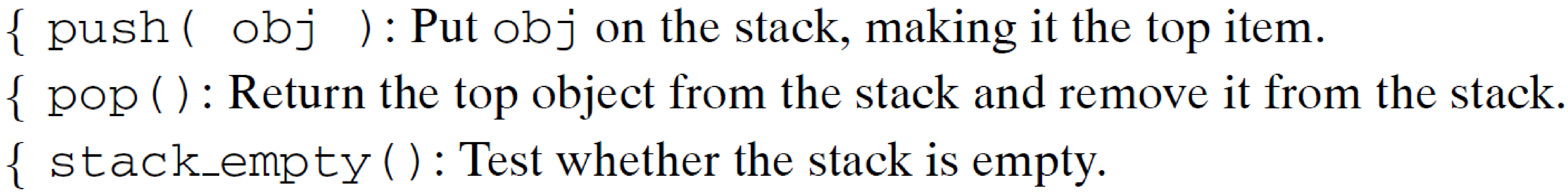
CONTENTS

1. STACK

Stacks

- They have a bottom and a top.
- We have access only to the top.
- Insertion (push) and deletion (pop) operations are performed on the top extreme.
- LIFO: Last In First Out.
- Applications:
  - Nested blocks, local variables, recursive definitions, or backtracking.
  - Parentheses and operator priorities.
  - Search in a labyrinth with backtracking.

Stack Operations



{ `push( obj )`: Put `obj` on the stack, making it the top item.
{ `pop()`: Return the top object from the stack and remove it from the stack.
{ `stack_empty()`: Test whether the stack is empty.

- We assume we want to store elements of type `item_t`.

Stack on an Infinite Array

- This is not realistic.
- `i` is the position where the next element will appear.

```
int i=0;
item_t stack[∞];

int stack_empty(void)
{    return( i == 0 );
}

void push( item_t x)
{    stack[i++] = x ;
}

item_t pop(void)
{    return( stack[ --i] );
}
```

Stack on a Finite Array

- `i` is the position where the next element will appear.
- We need to have a maximum size for the array.

```
int i=0;
item_t stack[MAXSIZE];

int stack_empty(void)
{   return( i == 0 );
}

int push( item_t x)
{   if ( i < MAXSIZE )
    {   stack[i++] = x ;  return( 0 );
    }
    else
       return( -1 );
}

item_t pop(void)
{   return( stack[ --i] );
}
```

Problems of Arrays

- They are of fixed size.
- The size needs to be decided in advance.
- The structure needs the full size, no matter how many items are really in the structure.

```
int i=0;
item_t stack[MAXSIZE];

int stack_empty(void)
{   return( i == 0 );
}

int push( item_t x)
{   if ( i < MAXSIZE )
    {   stack[i++] = x ;  return( 0 );
    }
    else
       return( -1 );
}

item_t pop(void)
{   return( stack[ --i] );
}
```

Underflow & Overflow

- We specify only overflow errors because they would not be present in an ideal implementation.
- We don't specify underflow errors because they are errors in the use of the data structure.
- However, we could also prevent them by checking whether the structure is empty.

```
int i=0;
item_t stack[MAXSIZE];

int stack_empty(void)
{   return( i == 0 );
}

int push( item_t x)
{   if ( i < MAXSIZE )
    {   stack[i++] = x ;   return( 0 );
    }
    else
       return( -1 );
}

item_t pop(void)
{   return( stack[ --i] );
}
```

Stack as Data Type

- We might need multiple stacks in the same program.
- Thus, we can create a stack data type that groups:
  - `st->base` is the array.
  - `st->top` is where the next item will be stored.
  - `st->size` is the size of the array.
- Then, we can create each stack dynamically.

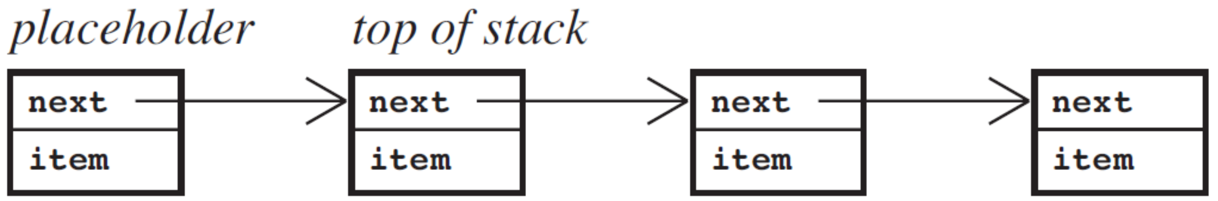
```
typedef struct {item_t *base; item_t *top;
                int size;} stack_t;

stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
                sizeof(item_t) );
    st->size = size;
    st->top = st->base;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->base == st->top );
}
```

Stack as Data Type

- `st->top` is the position where the next item will be stored.

```
int push( item_t x, stack_t *st)
{   if ( st->top < st->base + st->size )
    {  *(st->top) = x; st->top += 1; return( 0 );
    }
    else
       return( -1 );
}

item_t pop(stack_t *st)
{   st->top -= 1;
    return( *(st->top) );
}

item_t top_element(stack_t *st)
{   return( *(st->top -1) );
}

void remove_stack(stack_t *st)
{   free( st->base );
    free( st );
}
```

Stack as Linked List

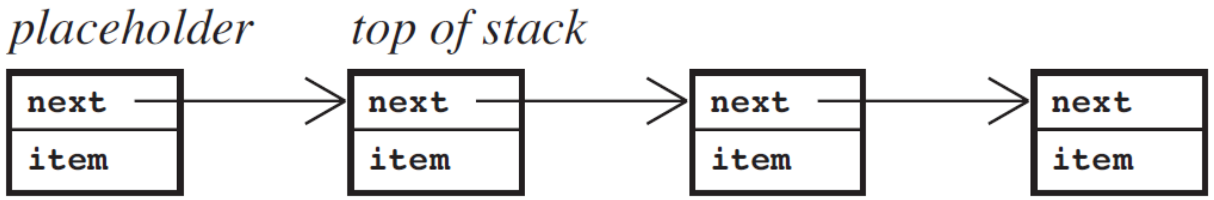


```
typedef struct st_t { item_t        item;
                      struct st_t *next; } stack_t;
```

- Each node has the item and a pointer to the next element.
- Each node can be stored anywhere in memory.
- The first node is a **placeholder**.
- It points to the top of the stack.
- We use the following functions:
    - `get_node()`: allocates memory for a new node.
    - `return_node(node)`: frees the memory allocated for node.

```
stack_t *create_stack(void)
{   stack_t *st;
    st = get_node();
    st->next = NULL;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->next == NULL );
}

void push( item_t x, stack_t *st)
{   stack_t *tmp;
    tmp = get_node();
    tmp->item = x;
    tmp->next = st->next;
    st->next = tmp;
}
```

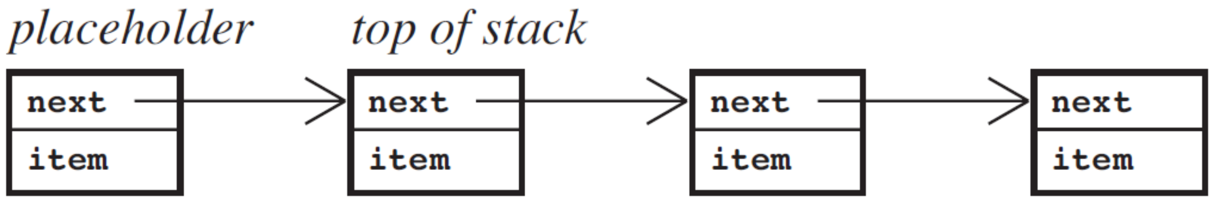
Stack as Linked List



- We use the following functions:
  - `get_node()`: allocates memory for a new node.
  - `return_node(node)`: frees the memory allocated for node.
- We use those rather than `malloc` and `free` because, at the end, we consider more efficient ways to manage memory.

```
item_t pop(stack_t *st)
{   stack_t *tmp; item_t tmp_item;
    tmp = st->next;
    st->next = tmp->next;
    tmp_item = tmp->item;
    return_node( tmp );
    return( tmp_item );
}

item_t top_element(stack_t *st)
{   return( st->next->item );
}

void remove_stack(stack_t *st)
{   stack_t *tmp;
    do
    {  tmp = st->next;
       return_node(st);
       st = tmp;
    }
    while ( tmp != NULL );
}
```

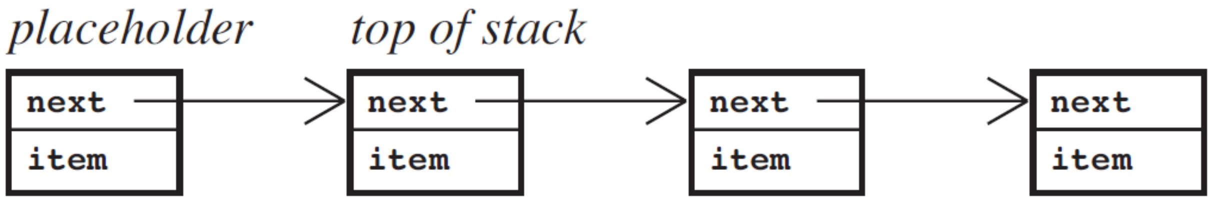
Advantages



- Frequently, it is preferable to implement the stack as a dynamically allocated structure in the form of linked list.
- This is because it is not of fixed size.
- We store only what we are using.
- There’s no need to handle overflow errors assuming the computer has unbounded memory.

```
item_t pop(stack_t *st)
{   stack_t *tmp; item_t tmp_item;
    tmp = st->next;
    st->next = tmp->next;
    tmp_item = tmp->item;
    return_node( tmp );
    return( tmp_item );
}

item_t top_element(stack_t *st)
{   return( st->next->item );
}

void remove_stack(stack_t *st)
{   stack_t *tmp;
    do
    {  tmp = st->next;
       return_node(st);
       st = tmp;
    }
    while ( tmp != NULL );
}
```

Disadvantages



- Possible decrease in speed.
- Dereferencing a pointer does not take longer than incrementing an index.
- But the location accessed by the pointer may be anywhere in memory, whereas the next component in an array is near.
- Arrays work well with the cache; dynamically linked lists might generate many cache misses.
- Thus, if you know the size, use arrays.

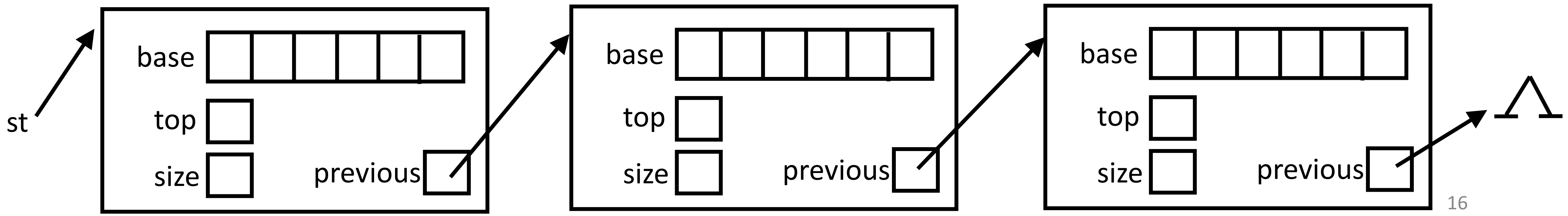
```
item_t pop(stack_t *st)
{    stack_t *tmp; item_t tmp_item;
     tmp = st->next;
     st->next = tmp->next;
     tmp_item = tmp->item;
     return_node( tmp );
     return( tmp_item );
}

item_t top_element(stack_t *st)
{    return( st->next->item );
}

void remove_stack(stack_t *st)
{    stack_t *tmp;
     do
     {  tmp = st->next;
        return_node(st);
        st = tmp;
     }
     while ( tmp != NULL );
}
```

Hybrid Implementation

```
typedef struct st_t { item_t *base;
                      item_t  *top;
                      int     size;
                   struct st_t *previous;} stack_t;
```

- If we want to combine these advantages, one could use a linked list of blocks.
- Each block contains an array.
- When the array becomes full, we link it to a new node with a new array (with previous).

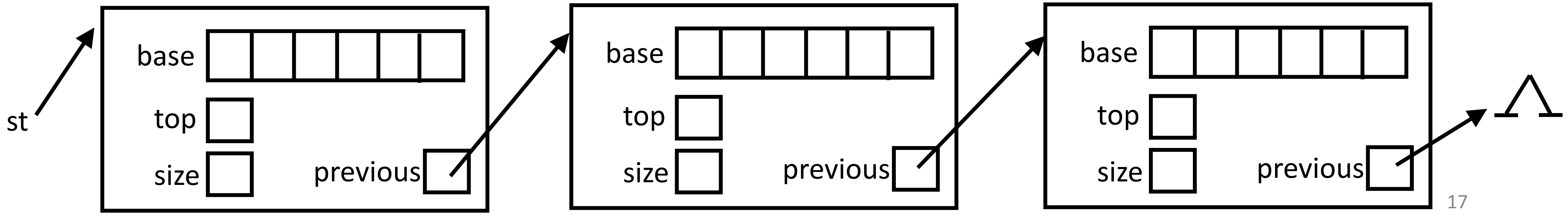
```
stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
                   sizeof(item_t) );
    st->size = size;
    st->top = st->base;
    st->previous = NULL;
    return( st );
}

int stack_empty(stack_t *st)
{   return( st->base == st->top &&
            st->previous == NULL);
}
```



Hybrid Implementation

- If we want to combine these advantages, one could use a linked list of blocks.
- Each block contains an array.
- When the array becomes full, we link it to a new node with a new array (with previous).

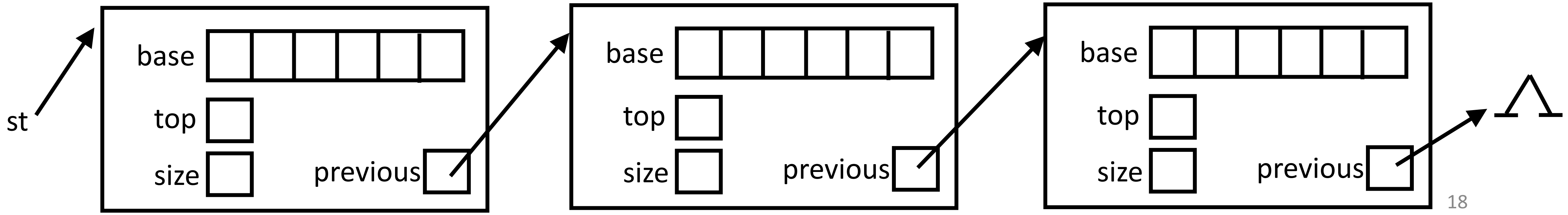
```
void push( item_t x, stack_t *st)
{   if ( st->top < st->base + st->size )
    {   *(st->top) = x; st->top += 1;
    }
    else
    {   stack_t *new;
        new = (stack_t *) malloc( sizeof(stack_t) );
        new->base = st->base;
        new->top = st->top;
        new->size = st->size;
        new->previous = st->previous;
        st->previous = new;
        st->base = (item_t *) malloc( st->size *
                   sizeof(item_t) );
        st->top = st->base+1;
        *(st->base) = x;
    }
}
```



Hybrid Implementation

- If we want to combine these advantages, one could use a linked list of blocks.
- Each block contains an array.
- When the array becomes full, we link it to a new node with a new array (with previous).

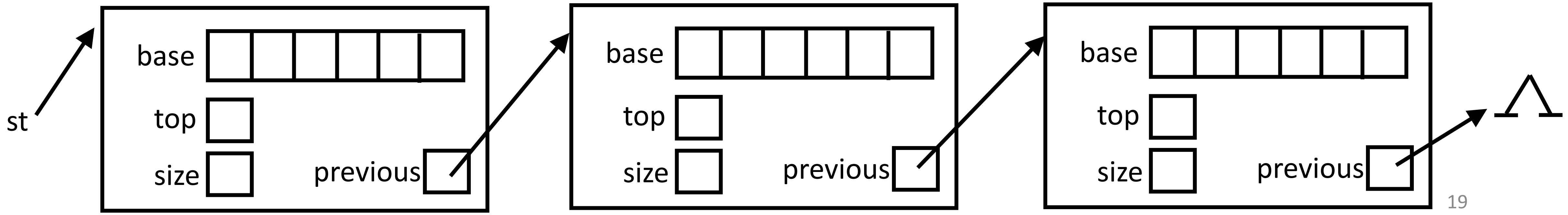
```
item_t pop(stack_t *st)
{   if( st->top == st->base )
    {  stack_t *old;
       old = st->previous;
       st->previous = old->previous;
       free( st->base );
       st->base = old->base;
       st->top = old->top;
       st->size = old->size;
       free( old );
    }
    st->top -= 1;
    return( *(st->top) );
}
```



Hybrid Implementation

- If we want to combine these advantages, one could use a linked list of blocks.
- Each block contains an array.
- When the array becomes full, we link it to a new node with a new array (with previous).

```
item_t top_element(stack_t *st)
{   if( st->top == st->base )
          return( *(st->previous->top -1) );
     else
           return( *(st->top -1) );
}

void remove_stack(stack_t *st)
{     stack_t *tmp;
      do
      {   tmp = st->previous;
          free( st->base );
          free( st );
          st = tmp;
      }
      while( st != NULL );
}
```

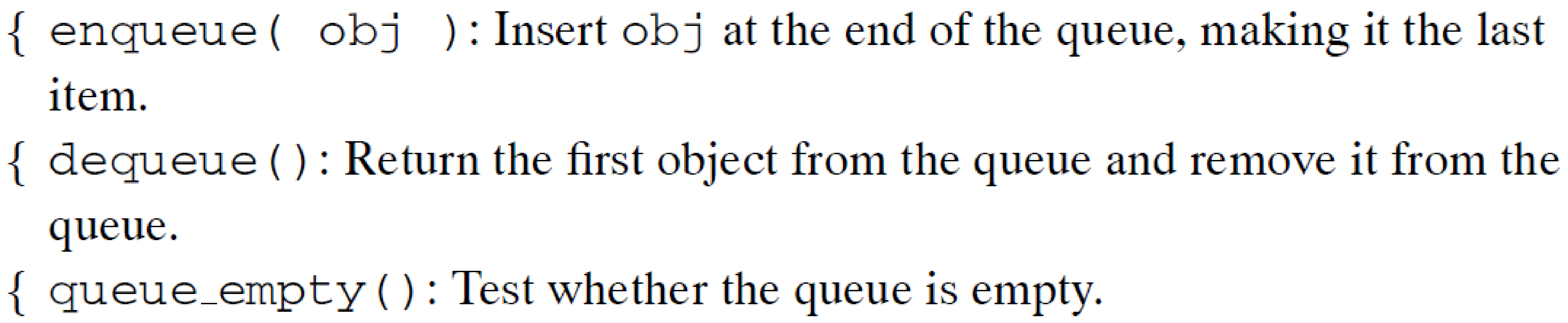


2. QUEUES

Queues

- They have a front and a rear.
- We have access only to the front.
- Insertion (enqueue) and deletion (dequeue) operations are performed on the rear and front extreme, respectively.
- FIFO: First In First Out.
- Applications:
  - Tasks that have to be processed cyclically.
  - Breadth First Search (BFS).

Queue Operations



{ `enqueue( obj )`: Insert `obj` at the end of the queue, making it the last item.

{ `dequeue()`: Return the first object from the queue and remove it from the queue.

{ `queue_empty()`: Test whether the queue is empty.

Queue on an Infinite Array

- This is not realistic.
- `upper` is the position where the next element will appear.
- `lower` is the position where the first element is.

```
int lower=0; int upper=0;
item_t queue[∞];

int queue_empty(void)
{   return( lower == upper );
}

void enqueue( item_t x)
{   queue[upper++] = x ;
}

item_t dequeue(void)
{   return( queue[ lower++] );
}
```

Queue on a Finite Array

```
typedef struct {item_t *base;
                int    front;
                int     rear;
                int     size;} queue_t;
```

- We need to use index calculation modulo the length of the array.
- Members:
  - `base`: array.
  - `front`: position of the first element.
  - `rear`: position of the next element.
  - `size`: size of the array.
- Empty Queue: `front==rear`.
- Full Queue: `front==(rear+2) mod size`.
  - Could it be `front==(rear+1) mod size`?

```
queue_t *create_queue(int size)
{   queue_t *qu;
    qu = (queue_t *) malloc( sizeof(queue_t) );
    qu->base = (item_t *) malloc( size *
                   sizeof(item_t) );
    qu->size = size;
    qu->front = qu->rear = 0;
    return( qu );
}

int queue_empty(queue_t *qu)
{   return( qu->front == qu->rear );
}
```

Queue on a Finite Array

- We need to use index calculation modulo the length of the array.
- Members:
  - `base`: array.
  - `front`: position of the first element.
  - `rear`: position of the next element.
  - `size`: size of the array.
- Empty Queue: `front==rear`.
- Full Queue: `front==(rear+2) mod size`.
  - Could it be `front==(rear+1) mod size`?

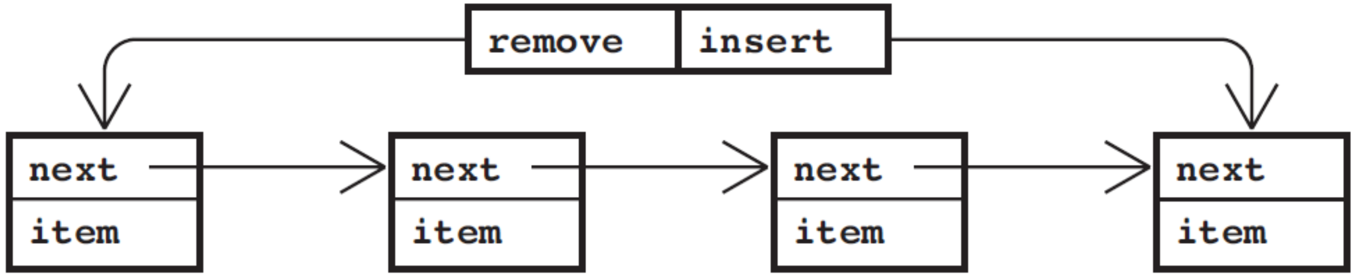
```
int enqueue( item_t x, queue_t *qu)
{   if ( qu->front != ((qu->rear +2)% qu->size) )
    {   qu->base[qu->rear] = x;
        qu->rear = ((qu->rear+1)%qu->size);
        return( 0 );
    }
    else
       return( -1 );
}

item_t dequeue(queue_t *qu)
{   int tmp;
    tmp = qu->front;
    qu->front = ((qu->front +1)%qu->size);
    return( qu->base[tmp] );
}
```

Queue on a Finite Array

- We need to use index calculation modulo the length of the array.
- Members:
  - `base`: array.
  - `front`: position of the first element.
  - `rear`: position of the next element.
  - `size`: size of the array.
- Empty Queue: `front==rear`.
- Full Queue: `front==(rear+2) mod size`.
  - Could it be `front==(rear+1) mod size`?

```
item_t front_element(queue_t *qu)
{   return( qu->base[qu->front] );
}

void remove_queue(queue_t *qu)
{   free( qu->base );
    free( qu );
}
```

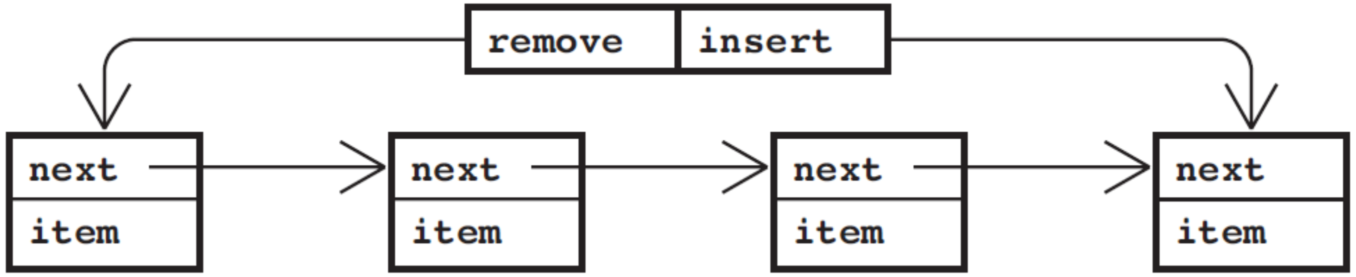
Queue as a Linked List



- We can implement the queue in a dynamic allocated fashion as a linked list.
- We can point the nodes from front to rear.
- We can also have a pointer to the front for dequeuing and a pointer to the rear for enqueueing:
    - `remove`
    - `insert`

```
typedef struct qu_n_t {item_t    item;
                        struct qu_n_t *next; } qu_node_t;
typedef struct {qu_node_t *remove;
                qu_node_t *insert; } queue_t;

queue_t *create_queue()
{   queue_t *qu;
    qu = (queue_t *) malloc( sizeof(queue_t) );
    qu->remove = qu->insert = NULL;
    return( qu );
}

int queue_empty(queue_t *qu)
{   return( qu->insert ==NULL );
}
```

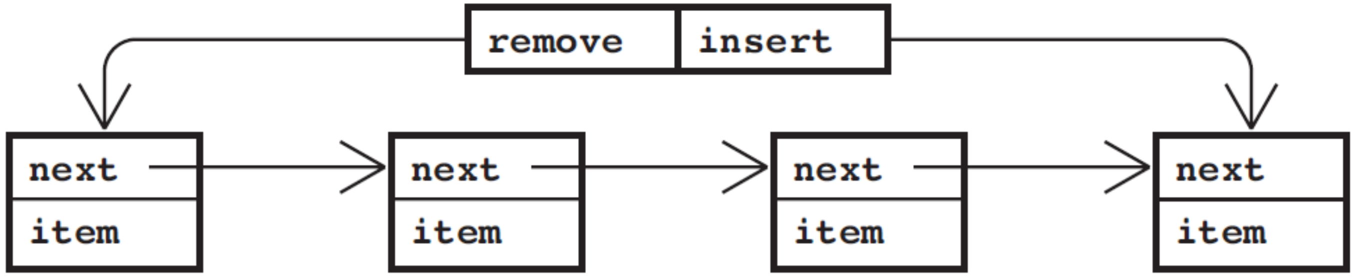
Queue as a Linked List



- We can implement the queue in a dynamic allocated fashion as a linked list.
- We can point the nodes from front to rear.
- We can also have a pointer to the front for dequeuing and a pointer to the rear for enqueueing:
    - `remove`
    - `insert`

```
void enqueue( item_t x, queue_t *qu)
{    qu_node_t *tmp;
     tmp = get_node();
     tmp->item = x;
     tmp->next = NULL; /* end marker */
     if ( qu->insert != NULL ) /* queue nonempty */
     {    qu->insert->next = tmp;
          qu->insert = tmp;
     }
     else /* insert in empty queue */
     {    qu->remove = qu->insert = tmp;
     }
}
```

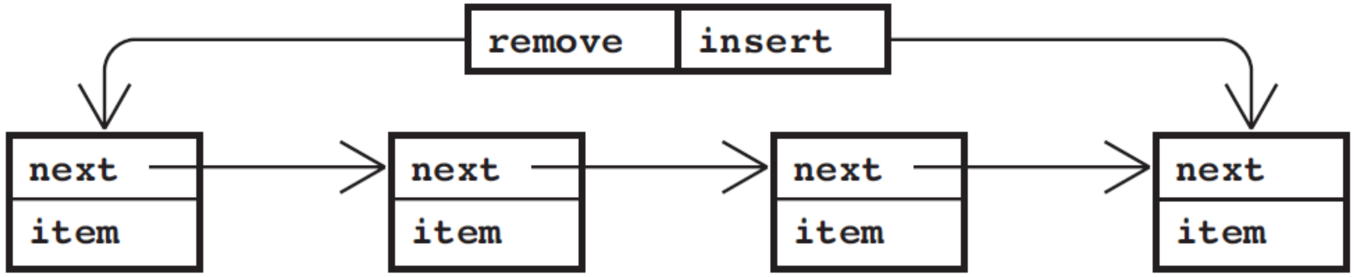
Queue as a Linked List



- We can implement the queue in a dynamic allocated fashion as a linked list.
- We can point the nodes from front to rear.
- We can also have a pointer to the front for dequeuing and a pointer to the rear for enqueueing:
    - `remove`
    - `insert`

```
item_t dequeue(queue_t *qu)
{   qu_node_t *tmp; item_t tmp_item;
    tmp = qu->remove; tmp_item = tmp->item;
    qu->remove = tmp->next;
    if( qu->remove == NULL ) /* reached end */
        qu->insert = NULL; /* make queue empty */
    return_node(tmp);

    return( tmp_item );
}
```

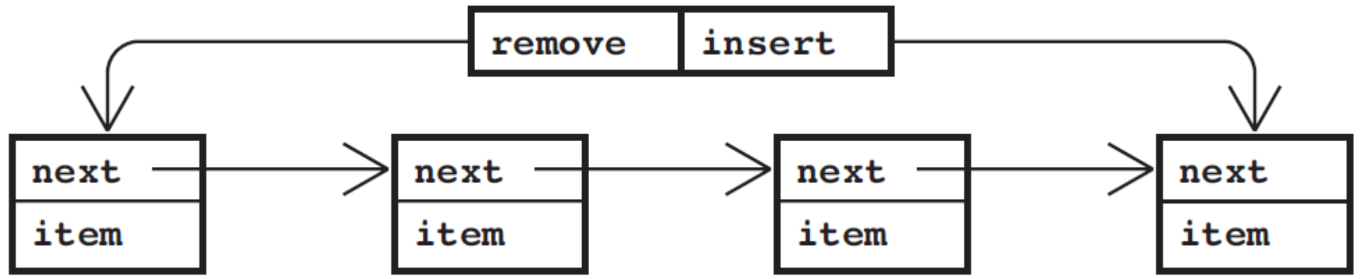
Queue as a Linked List



- We can implement the queue in a dynamic allocated fashion as a linked list.
- We can point the nodes from front to rear.
- We can also have a pointer to the front for dequeuing and a pointer to the rear for enqueueing:
  - `remove`
  - `insert`

```
item_t front_element(queue_t *qu)
{    return( qu->remove->item );
}

void remove_queue(queue_t *qu)
{    qu_node_t *tmp;
     while( qu->remove != NULL)
     { tmp = qu->remove;
       qu->remove = tmp->next;
       return_node(tmp);
     }
     free( qu );
}
```

Queue as a Linked List

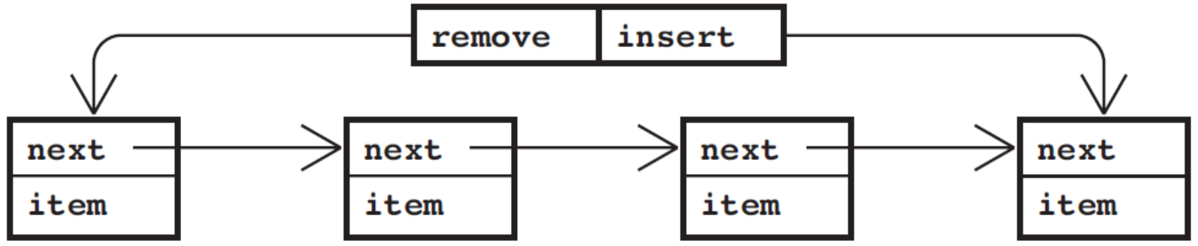


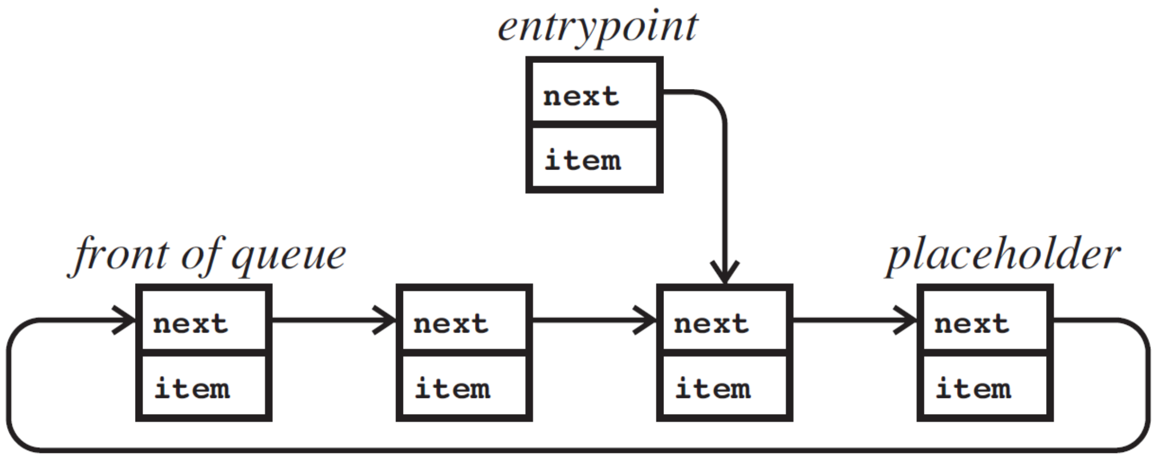
- We can implement the queue in a dynamic allocated fashion as a linked list.
- We can point the nodes from front to rear.
- We can also have a pointer to the front for dequeuing and a pointer to the rear for enqueueing:
  - `remove`
  - `insert`

```
item_t front_element(queue_t *qu)
{    return( qu->remove->item );
}

void remove_queue(queue_t *qu)
{    qu_node_t *tmp;
     while( qu->remove != NULL)
     { tmp = qu->remove;
       qu->remove = tmp->next;
       return_node(tmp);
     }
     free( qu );
}
```

Aesthetical Disadvantages

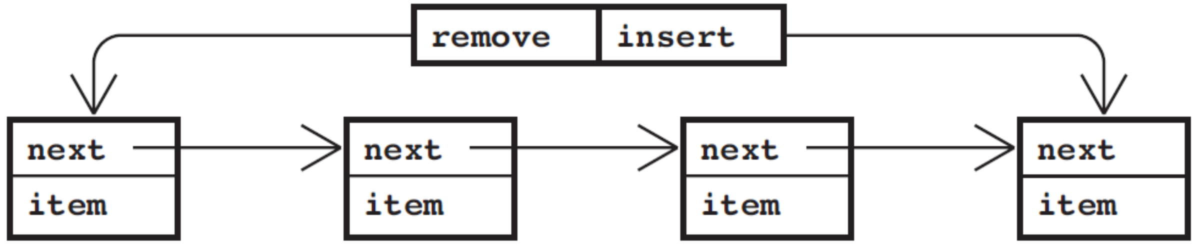
1. We need an entry point structure.
   - We can join the list together by making a cyclic list.
   - We don't need the remove pointer because the insertion point's next component points to the removal point.

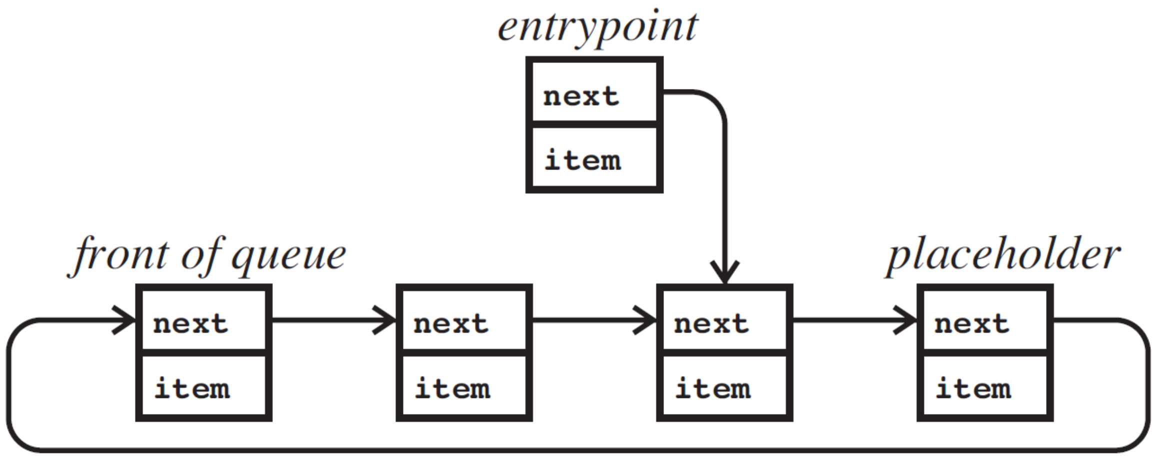




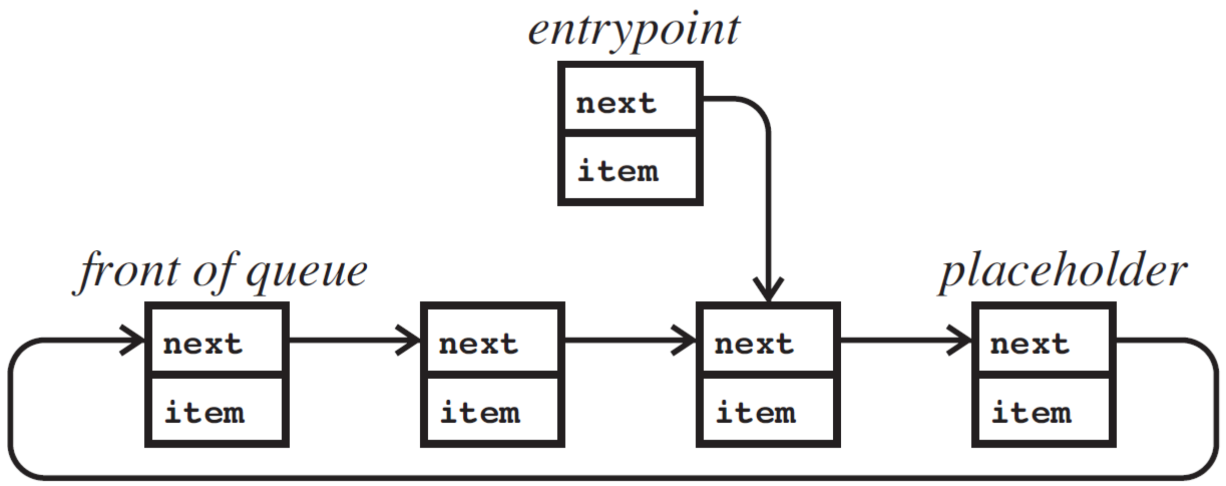
Aesthetical Disadvantages

2. We always need to treat the operations involving an empty queue differently.
   - We can insert a placeholder between the insertion point and the removal point.
   - The entry point still needs to point to the insertion end (or to the placeholder if the queue is empty).
   - At least for the insert, the empty list is no longer a special case. For deletion, it is not possible to dequeue in such case.





Queue as a Circular Linked List with Placeholder

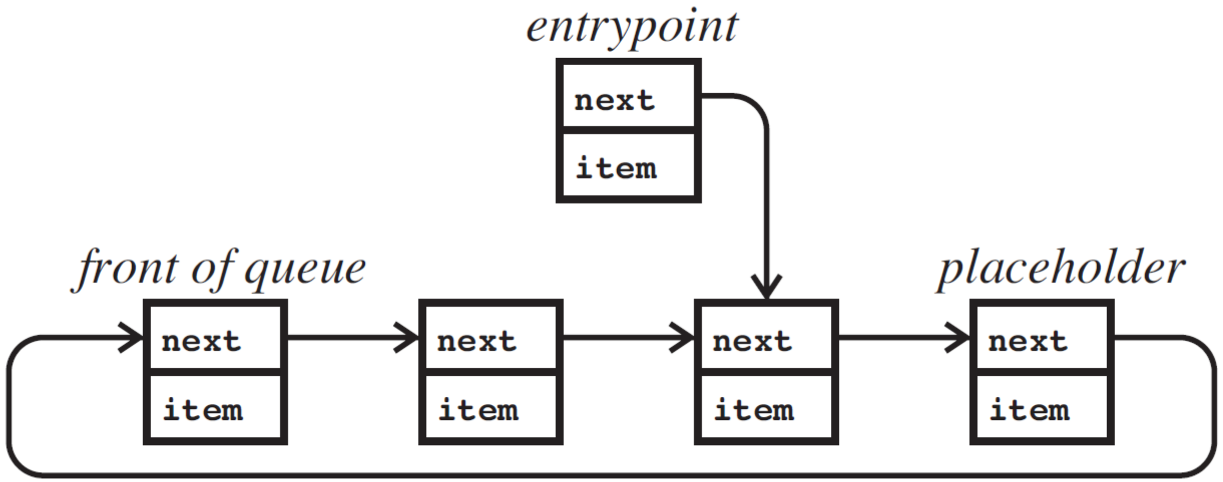


```
typedef struct qu_t { item_t  item;
                     struct qu_t *next; } queue_t;

queue_t *create_queue()
{    queue_t *entrypoint, *placeholder;
     entrypoint = (queue_t *) malloc( sizeof(queue_t) );
     placeholder = (queue_t *) malloc( sizeof(queue_t) );
     entrypoint->next = placeholder;
     placeholder->next = placeholder;
     return( entrypoint );
}

int queue_empty(queue_t *qu)
{    return( qu->next == qu->next->next );
}
```

- We have an entry point that points to the rear in the queue (or to the placeholder if it is empty).
- The placeholder separates the rear from the front.

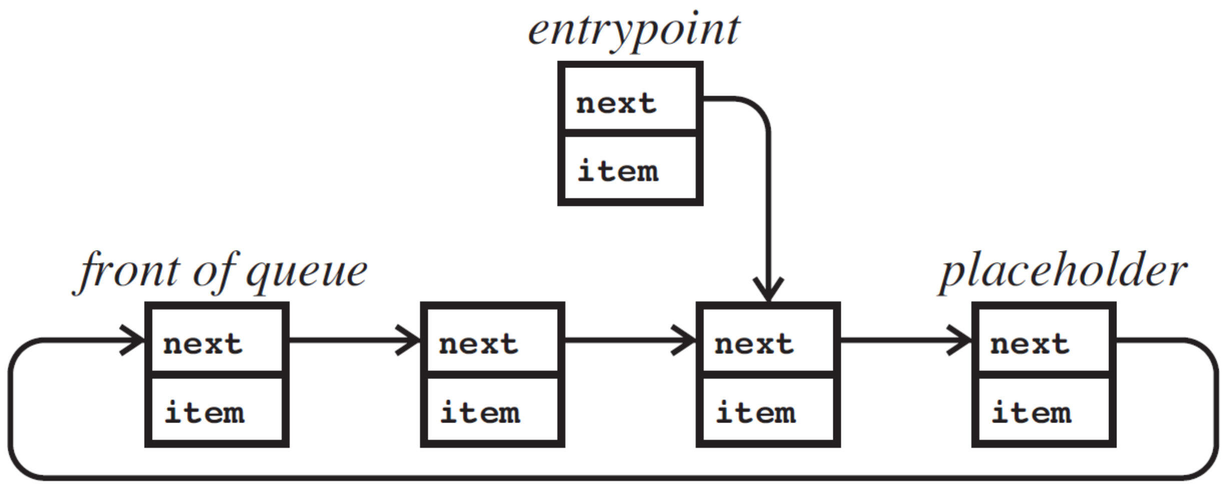
Queue as a Circular Linked List with Placeholder

```
void enqueue( item_t x, queue_t *qu)
{   queue_t *tmp, *new;
    new = get_node(); new->item = x;
    tmp = qu->next; qu->next = new;
    new->next = tmp->next; tmp->next = new;
}

item_t dequeue(queue_t *qu)
{   queue_t     *tmp;
    item_t  tmp_item;
    tmp = qu->next->next->next;
    qu->next->next->next = tmp->next;
    if( tmp == qu->next )
       qu->next = tmp->next;
    tmp_item = tmp->item;
    return_node( tmp );
    return( tmp_item );
}
```



- We have an entry point that points to the rear in the queue (or to the placeholder if it is empty).
- The placeholder separates the rear from the front.

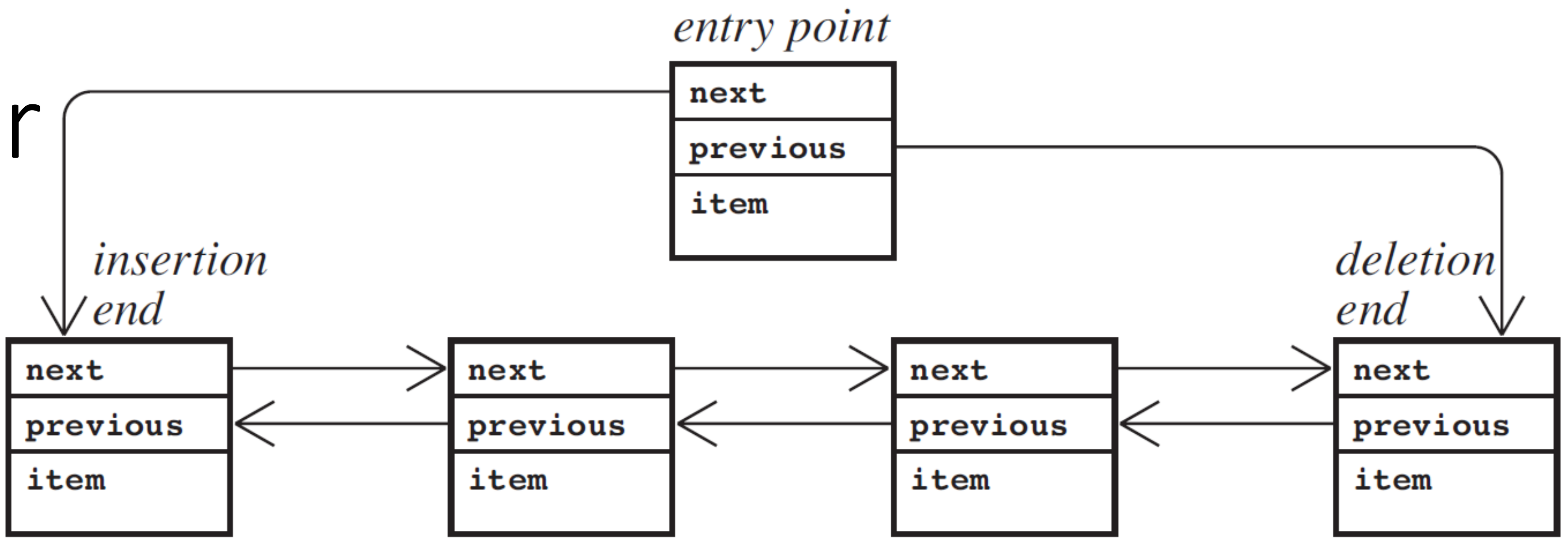
Queue as a Circular Linked List with Placeholder



```
item_t front_element(queue_t *qu)
{   return( qu->next->next->next->item );
}

void remove_queue(queue_t *qu)
{   queue_t *tmp;
    tmp = qu->next->next;
    while( tmp != qu->next )
    { qu->next->next = tmp->next;
      return_node( tmp );
        tmp = qu->next->next;
     }
    return_node( qu->next );
    return_node( qu );
  }
```

- We have an entry point that points to the rear in the queue (or to the placeholder if it is empty).
- The placeholder separates the rear from the front.

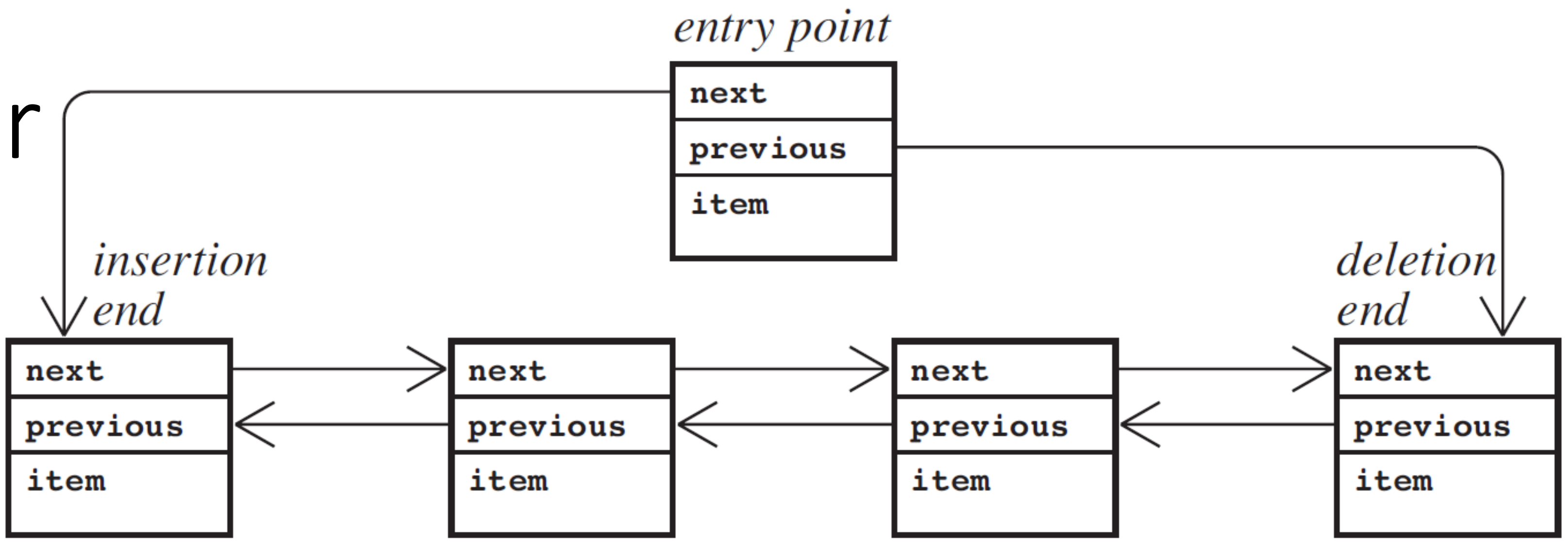
Queue as a Circular Doubly Linked List



```
typedef struct qu_t { item_t        item;
                     struct qu_t     *next;
                     struct qu_t *previous; } queue_t;

queue_t *create_queue()
{   queue_t *entrypoint;
    entrypoint = (queue_t *) malloc( sizeof(queue_t) );
    entrypoint->next = entrypoint;
    entrypoint->previous = entrypoint;
    return( entrypoint );
}
```

- With two pointers per node, no case distinctions are needed.
- But we need more space.
- We need extra work to keep the structure consistent.

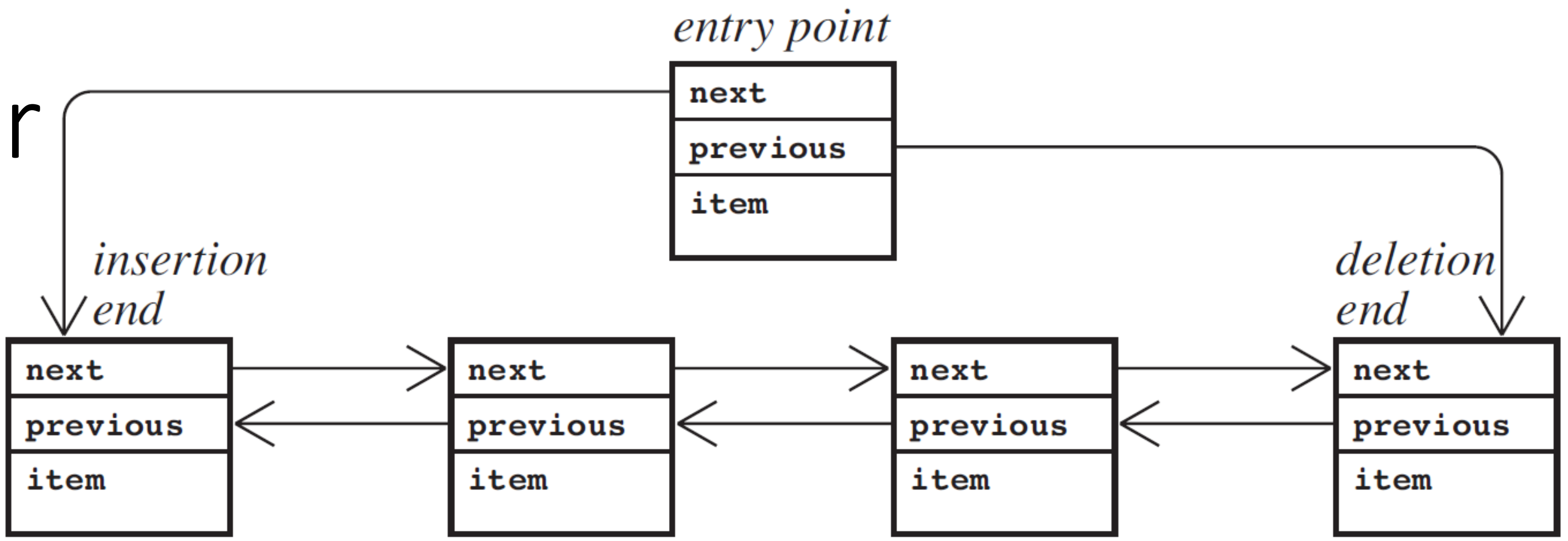
Queue as a Circular Doubly Linked List



```
int queue_empty(queue_t *qu)
{   return( qu->next == qu );
}

void enqueue( item_t x, queue_t *qu)
{   queue_t *new;
    new = get_node(); new->item = x;
    new->next = qu->next; qu->next = new;
    new->next->previous = new; new->previous = qu;
}
```

- With two pointers per node, no case distinctions are needed.
- But we need more space.
- We need extra work to keep the structure consistent.

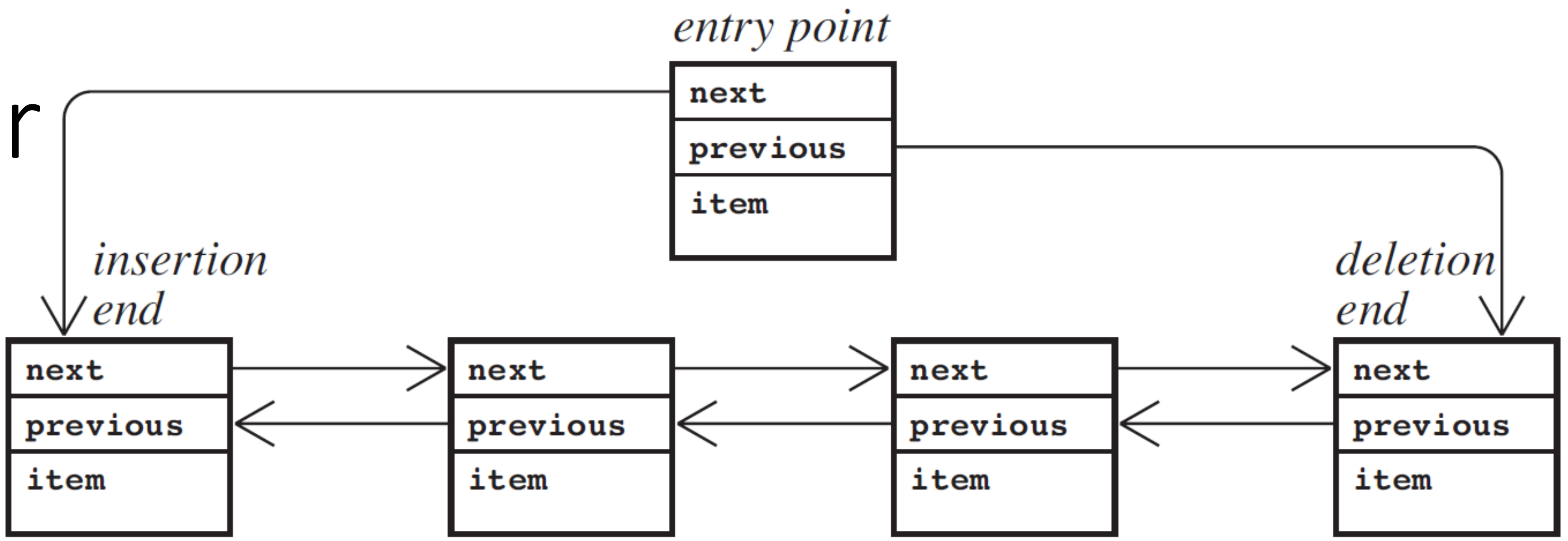
Queue as a Circular Doubly Linked List



```
item_t dequeue(queue_t *qu)
{   queue_t *tmp; item_t tmp_item;
    tmp = qu->previous; tmp_item = tmp->item;
    tmp->previous->next = qu;
    qu->previous = tmp->previous;
    return_node( tmp );
    return( tmp_item );
}
```

- With two pointers per node, no case distinctions are needed.
- But we need more space.
- We need extra work to keep the structure consistent.

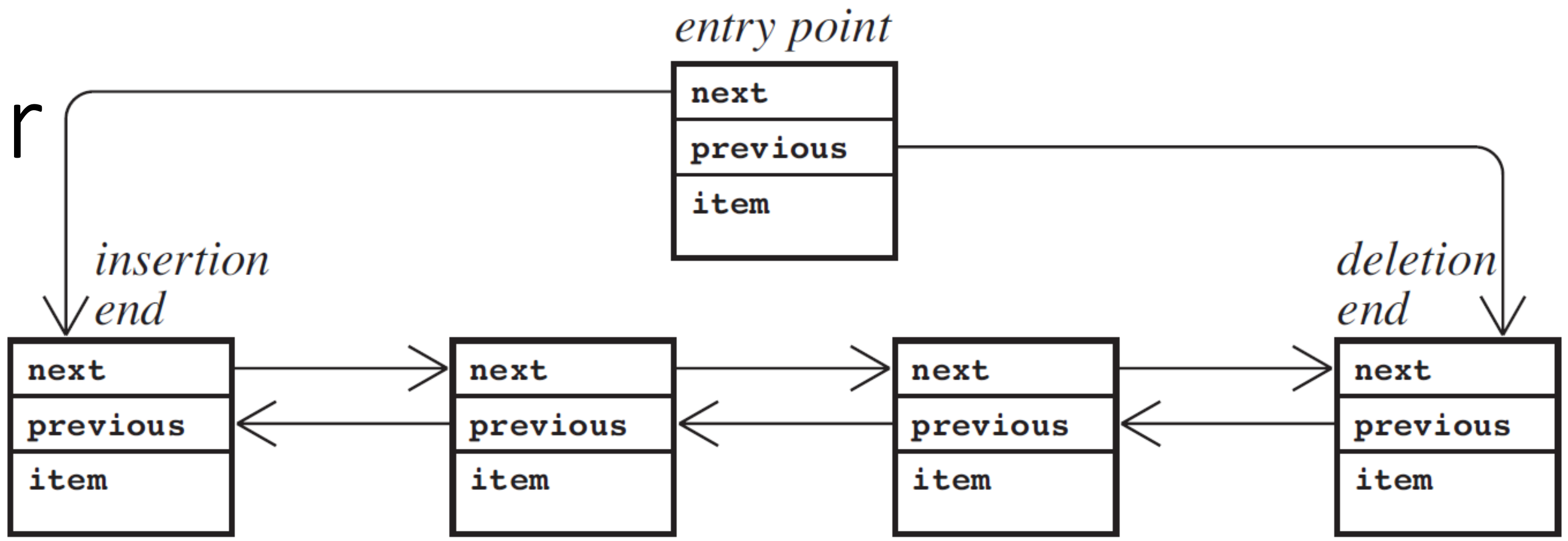
Queue as a Circular Doubly Linked List



```
item_t front_element(queue_t *qu)
{   return( qu->previous->item );
}
```

- With two pointers per node, no case distinctions are needed.
- But we need more space.
- We need extra work to keep the structure consistent.

Queue as a Circular Doubly Linked List



```
void remove_queue(queue_t *qu)
{   queue_t *tmp;
    qu->previous->next = NULL;
    do
    { tmp = qu->next;
      return_node( qu );
      qu = tmp;
    }
    while ( qu != NULL );

}
```

- With two pointers per node, no case distinctions are needed.
- But we need more space.
- We need extra work to keep the structure consistent.

Double-Ended Queue

- Generalization of stack and queue.
- It is a queue in which one can insert and delete at either end.
- Its implementation can be done as an array or as a doubly linked list.
- It does not have many applications.

3. DYNAMIC ALLOCATION OF NODES

Dynamic Allocation of Nodes through the OS Interface

- In a dynamically allocated structure, nodes are allocated and deallocated constantly.
- We could do so by using the operating-system interface functions `malloc` and `free`.
- However, they don't necessarily take constant time.
- Operating system memory allocation is ultimately the only way to get memory, but it is a complicated process.
- In any efficient implementation of a dynamically allocated structure, we cannot afford to access this operating-system-level memory management for every operation.

Dynamic Allocation of Nodes through an Intermediate Layer

- Instead, we introduce an intermediate layer.
- It only occasionally accesses operating-system-level memory management to get a large block of memory.
- Then, it gives out and receives back such memory, in small constant-sized pieces: the nodes.
- This layer includes the functions we presented in the previous sections:
    - `get_node()`: allocates memory for a node of a given type.
    - `return_node(node)`: deallocates memory of `node`.
- Those run in constant time.

Dynamic Allocation of Nodes through an Intermediate Layer

In particular, we will two collections of nodes to allocate nodes:

1. A stack `free_list`.
   - Comprised of the nodes that have been returned (from deletions).
   - When a new node needs to be allocated, we pop a node from this list.
2. An array block, called `*currentblock`, of `BLOCKSIZE` elements of the type of the node.
   - When a node needs to be allocated and `free_list` is empty, we allocate it from `*currentblock`.
   - In case it is empty, we allocate memory with `malloc` and then assign it one of its elements.

Intermediate Layer

```
typedef struct nd_t { struct nd_t *next;
                 /*and other components*/  } node_t;
#define BLOCKSIZE 256
node_t *currentblock = NULL;
int     size_left;
node_t *free_list = NULL;

node_t *get_node()
{ node_t *tmp;
  if( free_list != NULL )
  {   tmp = free_list;
      free_list = free_list -> next;
  }
  else
  {   if( currentblock == NULL || size_left == 0)
      {   currentblock =
                   (node_t *) malloc( BLOCKSIZE *
                                sizeof(node_t) );
          size_left = BLOCKSIZE;
      }
      tmp = currentblock++;
      size_left -= 1;
  }
  return( tmp );
}
```

```
void return_node(node_t *node)
{   node->next = free_list;
    free_list = node;
}
```

- Note that we never return memory to the system before the program ends.
- The amount of memory taken is the máximum amount taken by the data structure up to this moment.

Precautions in Dynamic Allocation

- Dynamic memory allocation is traditionally a source of many programming errors.
- It is hard to debug.
- A simple additional precaution to avoid some common errors is to add to the node another component: `int valid`.
- It can be filled with different values, depending on whether it has just been received back by `return_node(node)` or is given out by `get_node()`.

4. SHADOW COPIES OF ARRAY-BASED STRUCTURES

Shadow Copies of Array-Based Structures

- There is a systematic way to avoid the maximum-size problem of array-based structures.
- The program becomes less simple.
- We maintain two copies of the structure:
    - The currently active copy.
    - A larger-size structure that is under construction.
- We have to schedule the construction of the larger structure in such a way that it is finished and ready for use before the active copy reaches its maximum size.
- For this, we copy, in each operation of the old structure, a fixed number of elements from the old structure to the new structure.

Shadow Copies of Array-Based Structures

- When the content of the old structure is completely copied into the new larger structure, the old structure is removed and the new is taken as the active structure.
- This is also done when there is a deletion that yields the situation of the old structure being completely copied into the new one.
- When necessary, construction of an even larger structure is begun.
- The program is still simple.
- It only introduces a constant overhead.
- However, whenever there are changes in the active structure, they must also be applied in the structure under construction.

Shadow Copies of Array-Based Structures

The connection between copying threshold size, new maximum size, and the number of items copied is as follows:

- If the current structure has maximum size s_{max}, and
- we begin copying as soon as its actual size has reached αs_{max} , with $\frac{1}{2} \leq \alpha$,
- the new structure has maximum size $2s_{max}$, and
- each operation increases the actual size by at most 1,

Then, there are at least $(1 - \alpha)s_{max}$ steps left to complete the copying of at most s_{max} elements in the structure.

We need to copy $\left\lceil \frac{s_{max}}{(1-\alpha)s_{max}} \right\rceil = \left\lceil \frac{1}{1-\alpha} \right\rceil$ elements in each operation.

Shadow Copies of Array-Based Structures

- We doubled the maximum size when creating the new structure.
- But we could have chosen any size βs_{max}, $\beta > 1$, as long as $\alpha\beta > 1$.
- Otherwise, we would have to start copying again before the previous copying process was finished.
- In the next slide, we see a particular example with $\alpha = 0.75$.

Shadow Copies of Array-Based Structures

The connection between copying threshold size, new maximum size, and the number of items copied is as follows:

- If the current structure has maximum size s_{max}, and
- we begin copying as soon as its actual size has reached $0.75s_{max}$.
- the new structure has maximum size $2s_{max}$, and
- each operation increases the actual size by at most 1,

Then, there are at least $0.25s_{max}$ steps left to complete the copying of at most s_{max} elements in the structure.

We need to copy $\left\lceil \frac{1}{1-\alpha} \right\rceil = \left\lceil \frac{1}{0.25} \right\rceil = 4$ elements in each operation.

Shadow Copies of Array-Based Stack

```
typedef struct { item_t  *base;
                 int       size;
                 int   max_size;
                 item_t  *copy;
                 int copy_size; }     stack_t;

stack_t *create_stack(int size)
{   stack_t *st;
    st = (stack_t *) malloc( sizeof(stack_t) );
    st->base = (item_t *) malloc( size *
                     sizeof(item_t) );
    st->max_size = size;
    st->size = 0; st->copy = NULL; st->copy_size = 0;
    return( st );
}
```

Shadow Copies of Array-Based Stack

```
int stack_empty(stack_t *st)
{   return( st->size == 0);
}

void push( item_t x, stack_t *st)
{   *(st->base + st->size) = x;
    st->size += 1;
    if ( st->copy != NULL ||
    st->size >= 0.75*st->max_size )
    {  /* have to continue or start copying */
       int additional_copies = 4;
       if( st->copy == NULL )
       /* start copying: allocate space */
       {  st->copy =
          (item_t *) malloc( 2 * st->max_size *
           sizeof(item_t) );
       }
       /* continue copying: at most 4 items
          per push operation */
       while( additional_copies > 0 &&
              st->copy_size < st->size )
       {  *(st->copy + st->copy_size) =
                            *(st->base + st->copy_size);
          st->copy_size += 1; additional_copies -= 1;
       }
       if( st->copy_size == st->size)
       /* copy complete */
       {  free( st->base );
          st->base = st-> copy;
          st->max_size *= 2;
          st->copy = NULL;
          st->copy_size = 0;
       }
    }
}
```

Shadow Copies of Array-Based Stack

```
item_t pop(stack_t *st)
{   item_t tmp_item;
    st->size -= 1;
    tmp_item = *(st->base + st->size);
    if( st->copy_size == st->size) /* copy complete */
    {  free( st->base );
       st->base = st-> copy;
       st->max_size *= 2;
       st->copy = NULL;
       st->copy_size = 0;
    }
    return( tmp_item );
}
```

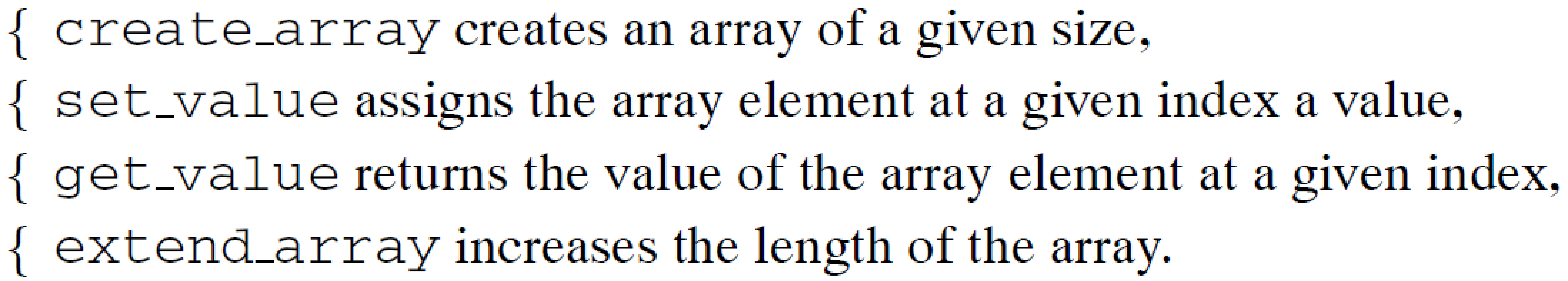
```
item_t top_element(stack_t *st)
{   return( *(st->base + st->size - 1) );
}

void remove_stack(stack_t *st)
{   free( st->base );
    if( st->copy != NULL )
       free( st->copy );
    free( st );
}
```

Normal Array

- Normal arrays need to be declared of a fixed size.
- They are allocated somewhere in memory.
- The space that is reserved cannot be increase because it might conflict with space allocated for other variables.
- Access to an array element is fast: it is just one address computation.

Extendible Array

- Some systems support a different type of array, which can be made larger.
- Accessing an element is more complicated and it is really an operation of a nontrivial data structure.
- This structure supports the following operations:



{ `create_array` creates an array of a given size,

{ `set_value` assigns the array element at a given index a value,

{ `get_value` returns the value of the array element at a given index,

{ `extend_array` increases the length of the array.

Extendible Array

- To implement this structure, we use the same technique of building shadow copies.
- The difference is that this structure does not just grow by a single item in each operation.
- The function `extend_array` can make it much larger in a single operation.
- Still, we can achieve an amortized constant time per operation.

Extendible Array

- When an array of size s is created, we allocate more space than requested.
- In particular, we always allocate is always a power of 2.
- We initially allocate an array of size $2^{\lceil \lg s \rceil}$ and store the start position of the array.
- We also store the current and maximum size in a structure that identifies the array.
- Each time `extend_array` is performed, we need to check whether the maximum size is larger than the requested size.
  - In that case, we just increase the current size.
  - Otherwise, we have to create another array whose size is the next number 2^k that is greater than the requested size. And of course, copy all elements.

Complexity of Extendible Array Operations

- Accessing an array element is always done in $O(1)$ time.
- Extending the array can take linear time in the size of the array.
- But the amortized complexity is not that bad:
- If the ultimate size of the array is $2^{\lceil \lg k \rceil}$, then we would have to have copied arrays of size $1,2,4,\ldots, 2^{\lceil \lg k \rceil - 1}$.
- So, we spend a total time of $O(k)$ with `extend_array` because

$$\sum_{i=1}^{\lceil \lg k \rceil - 1} 2^i = \frac{2^{\lceil \lg k \rceil} - 1}{1} = k - 1$$

- Then, we spend $O(1)$ with each `extend_array` operation that did not copy the array.

Complexity of Extendible Array Operations

- **Theorem:** An extendible array structure with shadow copies performs any sequence of n `set_value`, `get_value`, and `extend_array` operations on an array whose final size is k in time $O(n+k)$.

- If we assume that each element of the array is accessed at least once, so that the final size is at most the number of elements access operations, this gives an amortized $O(1)$ time per operation.

Problems with Extendible Array

- It would be natural to distribute the copying of the elements over the later access operations; however, it is not possible because we don't control over the `extend_array` operations.

- Pointers to array elements are different from normal pointers because the position of the array can change.

- Thus, in general, extendible arrays should be avoided, even if the language supports them.

BIBLIOGRAPHY

- Peter Brass. Advanced Data Structures. Cambridge University Press. 2008.