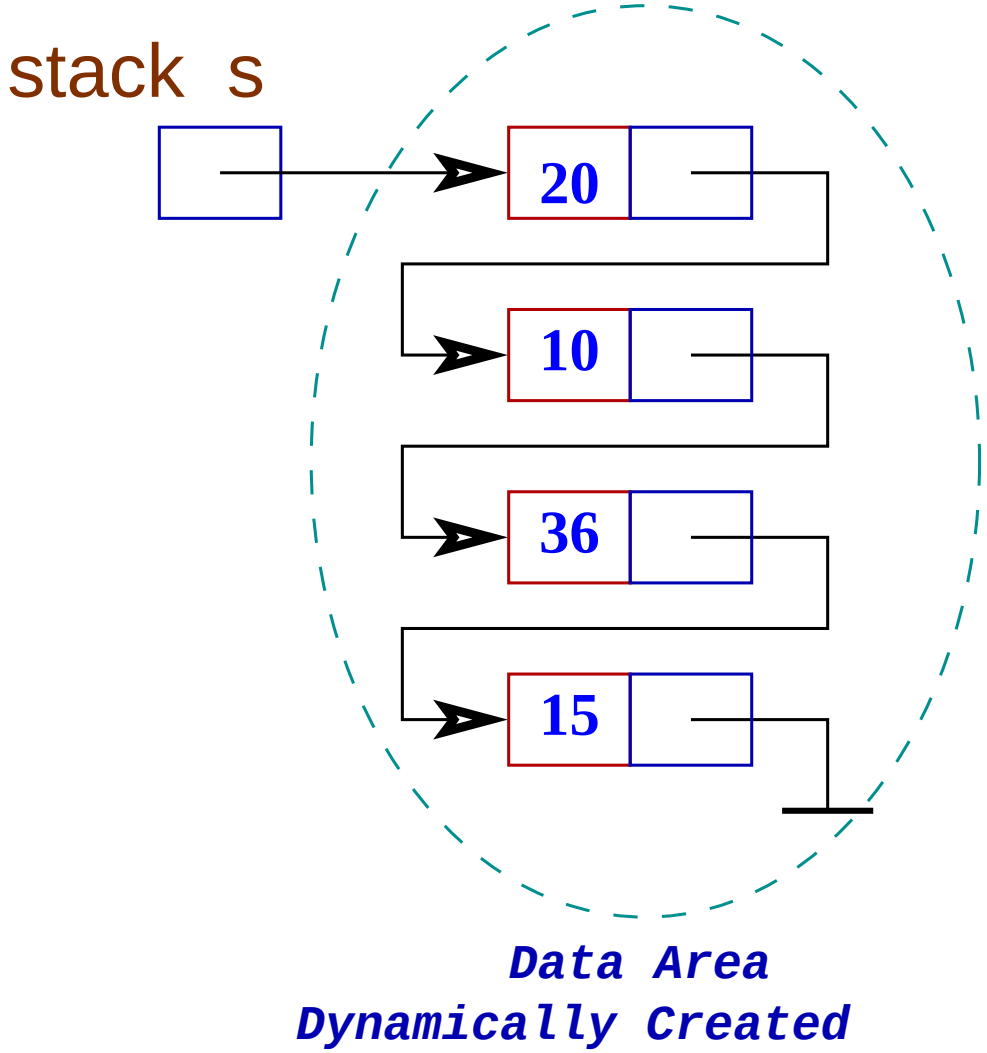
Stack & Queue on Self-Referencing Structures

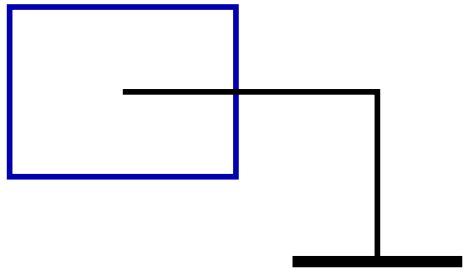
Representation of Stack

```
struct stack {
        int data ;
        struct stack *next ;
};
typedef struct stackNode node, *stack ;
```


stack s
20
10
36
15
Data Area
Dynamically Created

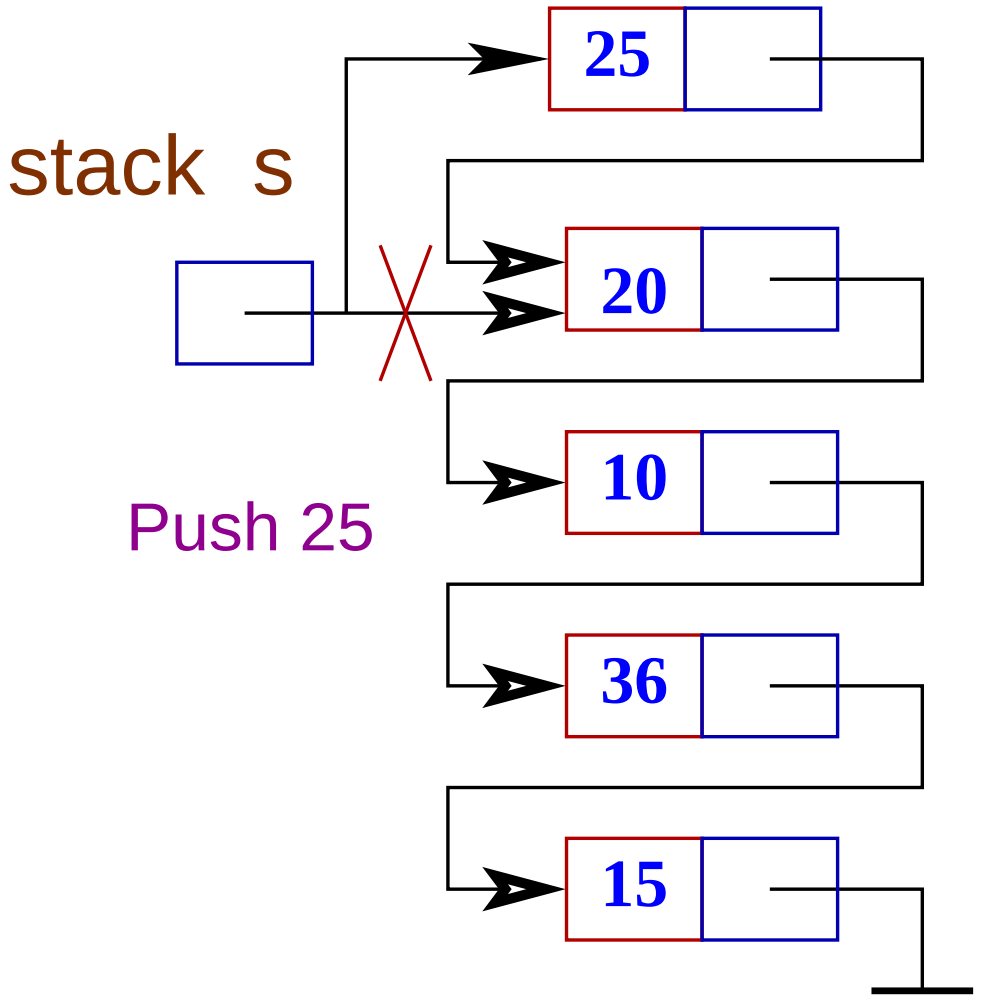
Empty Stack

stack s

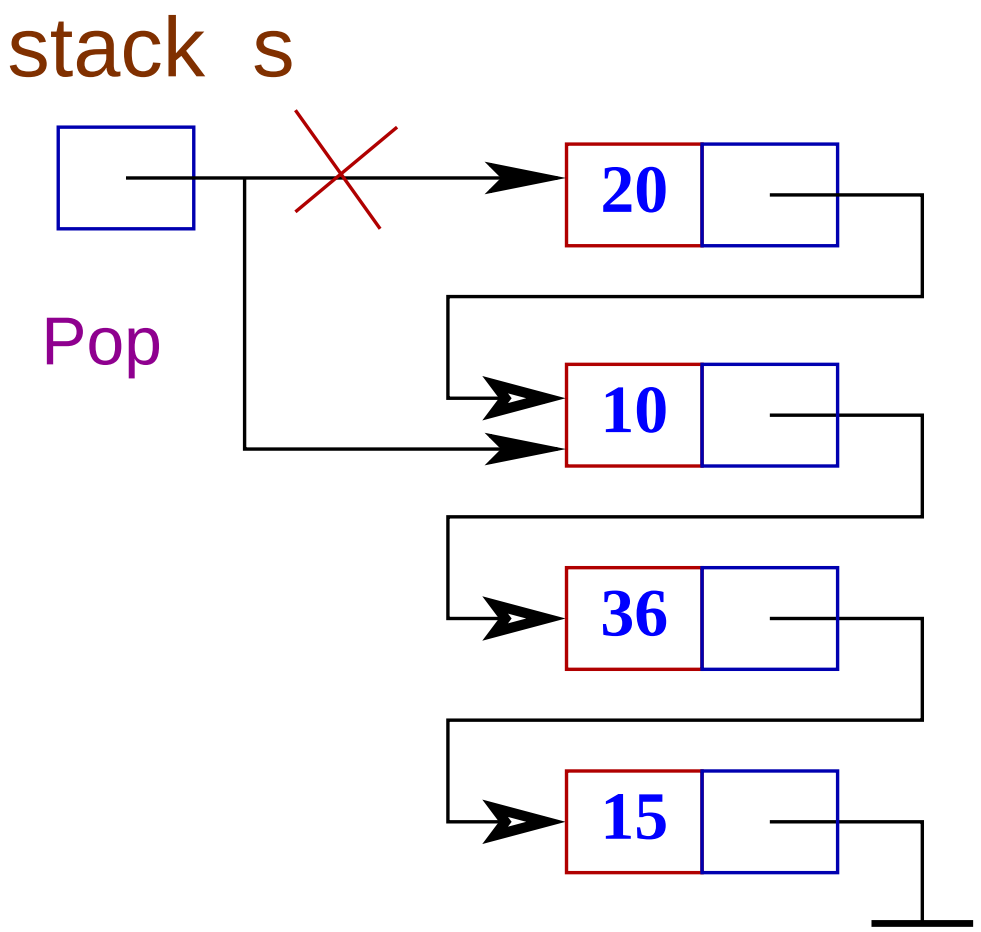


Empty Stack

Push



Pop



Interface File: stackSR.h

```
#ifndef _STACKSR_H
#define _STACKSR_H

#include <stdio.h>
#include <stdlib.h>
#define ERROR 1
#define OK    0

struct stack { // stackSR.h
       int data ;
       struct stack *next ;
} ;
```

```
typedef struct stack node, *stack ;

#define INIT(s) ((s)=NULL)
#define ISEMPTY(s)  ((s) == NULL)
int push(stack * , int) ;
int pop(stack *) ;
int top(stack, int *) ;

#endif
```

Note

We macro define INIT and ISEMPTY as the code is very simple. We also could have done inline function.

Implementation File: stackSR.c

```
#include "stackSR.h"
int push(stack *s, int n) { // stackSR.c
    stack temp ;

    temp=(stack)malloc(sizeof(node)) ;
    if(temp == NULL) {
       printf("The STACK is full\n");
       return ERROR ;
    }
    temp->data=n;
    temp->next=*s ;
    *s = temp ;
```

```
    return OK ;
}

int pop(stack *s) {
    stack temp ;
    if(ISEMPTY(*s)) {
       printf("The STACK is empty\n");
       return ERROR ;
    }
    temp=*s;
    *s=(*s)->next ;
    free(temp) ;
    return OK ;
}
```

```
int top(stack s , int *val) {
    if(ISEMPTY(s)) {
       printf("The STACK is empty\n") ;
       return ERROR ;
    }
    *val = s -> data ;
    return OK ;
}
```

User Program: testStackSR.c

```
#include <stdio.h>
#include "stackSR.h"

int main() // testStackSR.c
{
    stack s ;
    int x , err , val ;
    char c ;

    INIT(s);
    printf(" 'U' for push (U 15)\n 'O' for pop\n 'T' fo
    printf(" 'E' for exit :\n");
```

```
while((c = getchar()) != 'e' && c != 'E')
        switch(c) {
    case 'u' :
    case 'U' :
                    scanf("%d",&x);
                    err = push(&s,x);
                    break;
    case 'o' :
    case 'O' :
                    err = pop(&s);
                    break;
    case 't' :
    case 'T' :
                    err = top(s , &val) ;
```

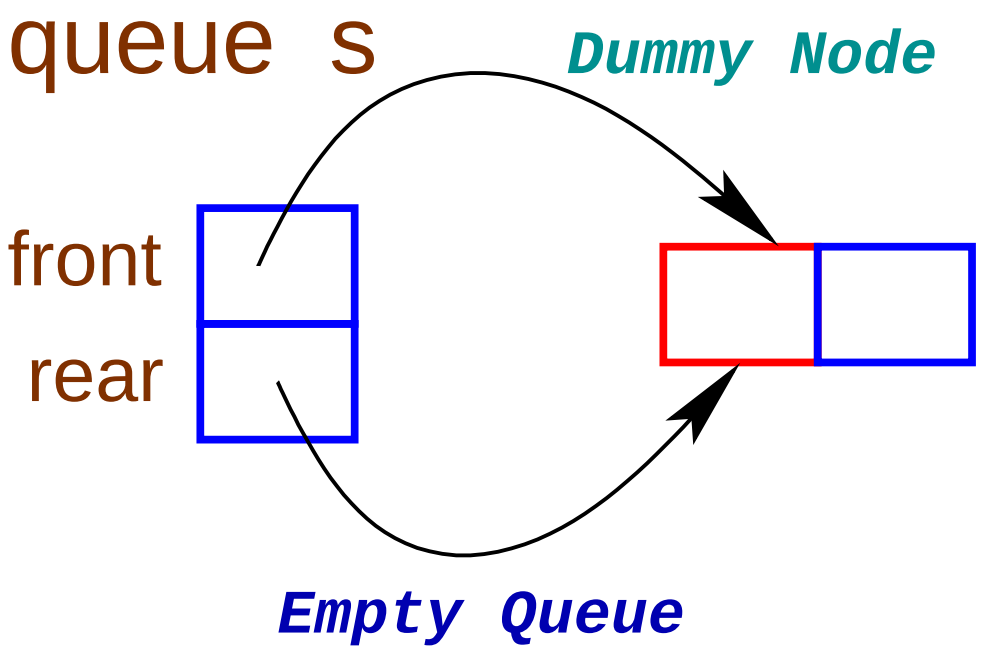
```
                    if(!err) printf("%d\n", val);
                    break;
        case '\n' :
        case '\t' :
        case ' '  :
                    break;
        default :
                    printf("Token Unknown\n");
            }
    return 0;
}
```

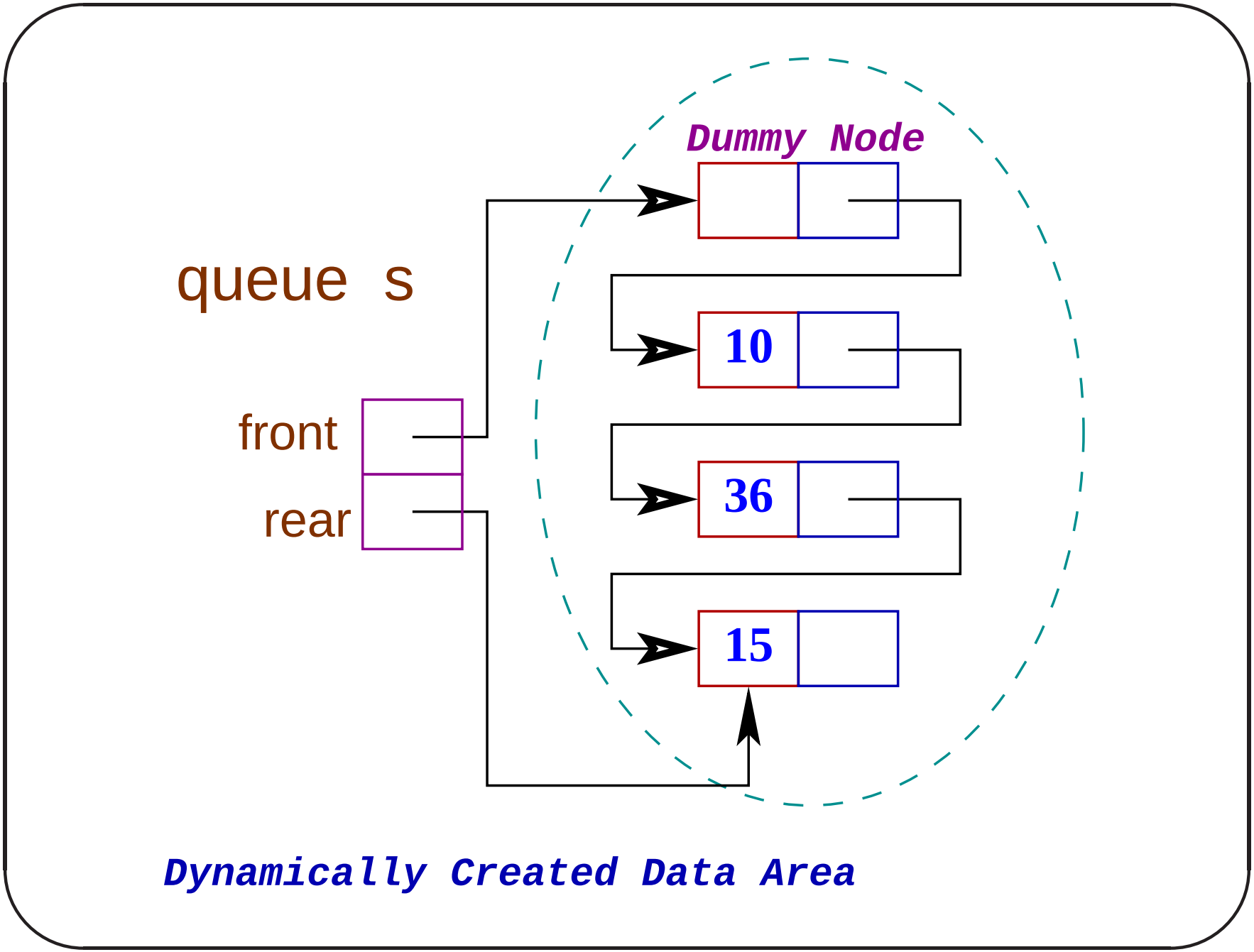
Representation of Queue

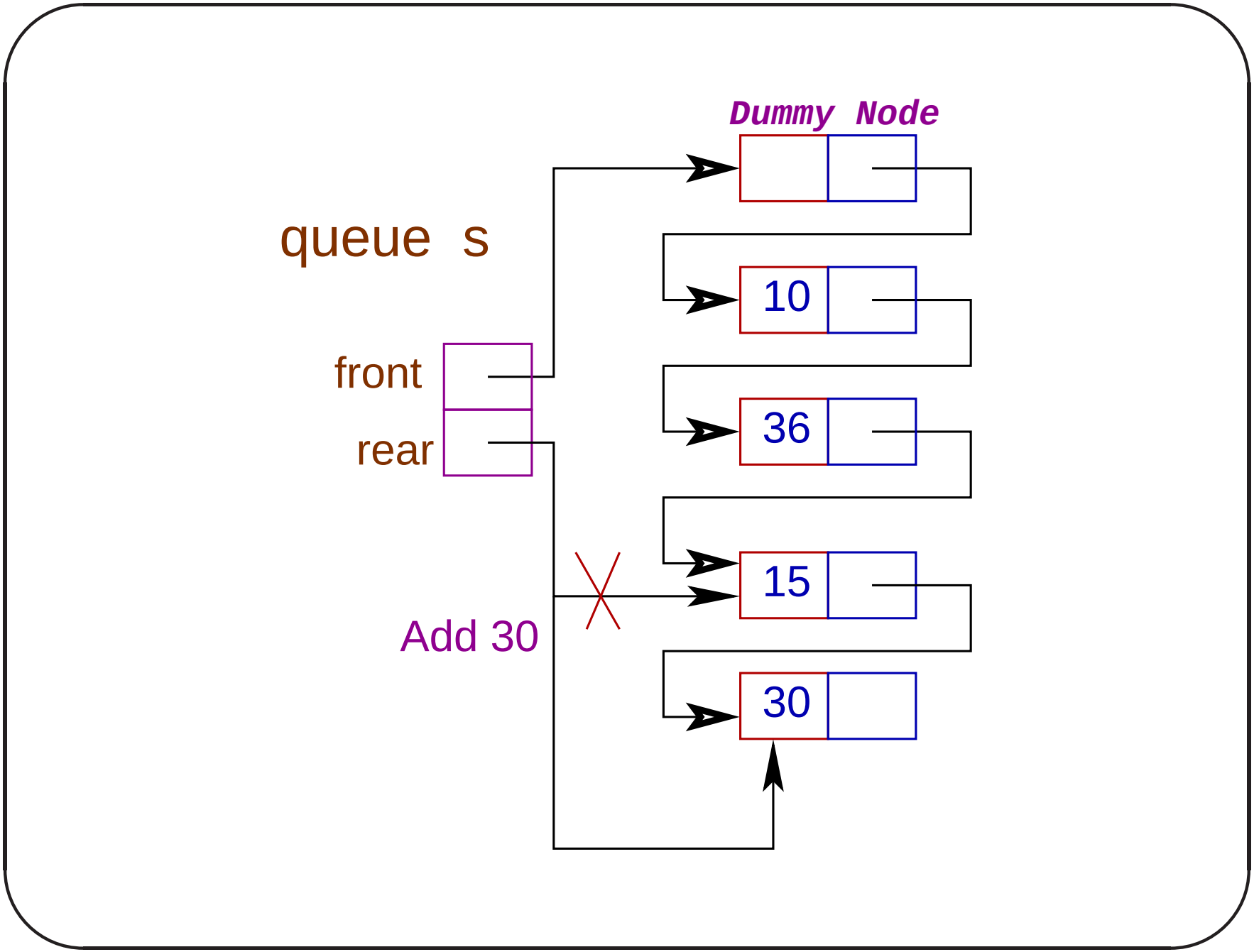
```
struct queue {
        int data ;
        struct queue *next ;
};
typedef struct {
         struct queue *front, *rear ;
} queue ;
```

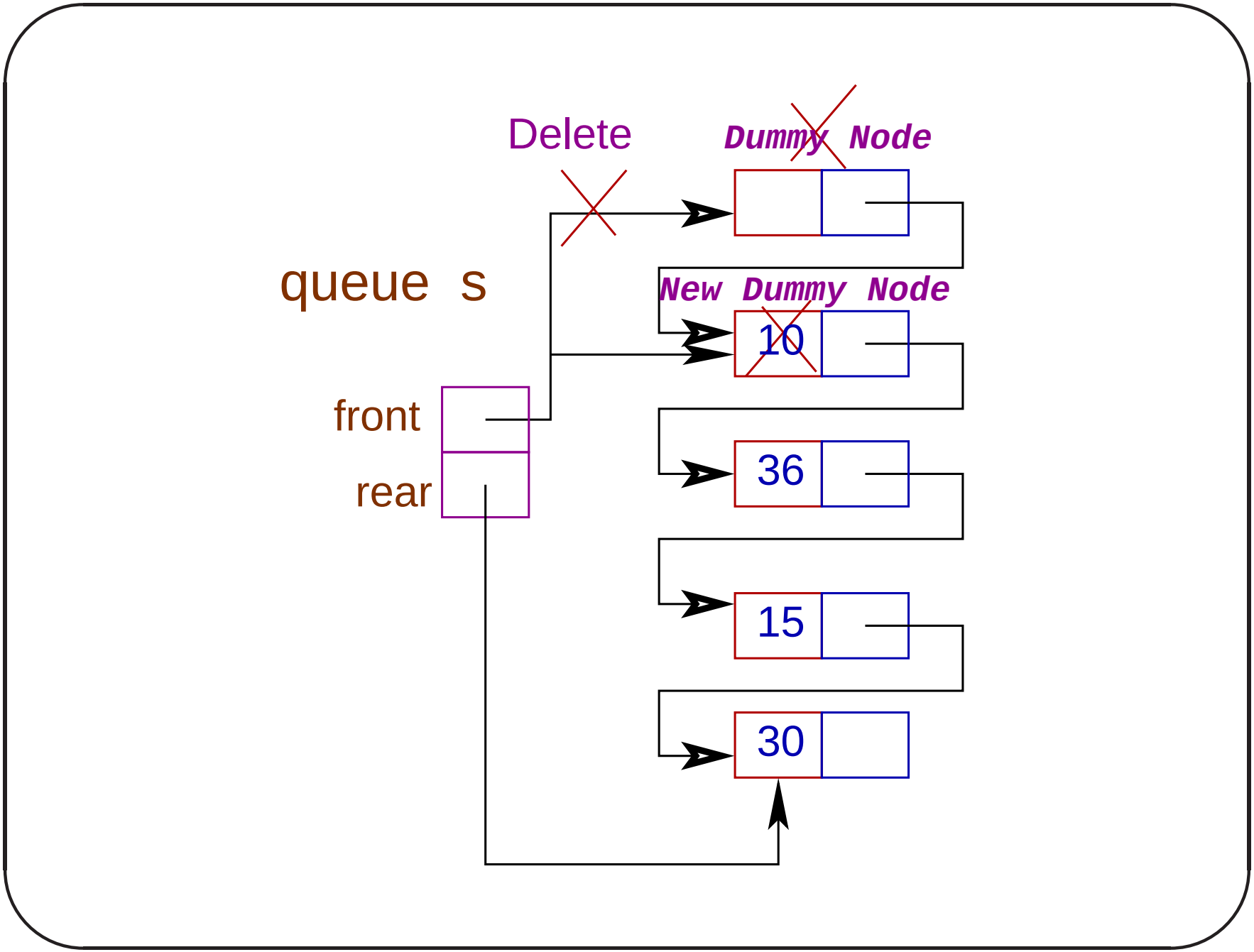
Note

We use a dummy node. In an empty queue, both the `front` and the `rear` pointers points to the dummy node.


queue s
Dummy Node
front
rear
Empty Queue


Dummy Node
queue s
front
rear
10
36
15
Dynamically Created Data Area


Dummy Node
queue s
front
rear
10
36
15
30
Add 30


Delete
Dummy Node
queue s
New Dummy Node
10
front
36
rear
15
30

Interface File: queueSR.h

```
#ifndef _QUEUESR_H
#define _QUEUESR_H

#include <stdio.h>
#include <stdlib.h>
#define ERROR 1
#define OK 0

struct queue { // queueSR.h
       int data ;
       struct queue *next ;
};
```

```
typedef struct queue node;
typedef struct {
        struct queue *front, *rear ;
} queue ;

#define ISEMPTY(q) ((q).front == (q).rear)
void init(queue *) ;
int add(queue *, int) ;
int delete(queue *);
int front(queue, int *) ;
#endif
```

Implementation File: queueSR.c

```
#include "queueSR.h"
void init(queue *qP) { // queueSR.c
     node *temp ;
     temp = (node *)malloc(sizeof(node)) ;
     qP->front=qP->rear=temp ;
}

int add(queue *qP, int n) {
    node *temp ;

    temp=(node *)malloc(sizeof(node)) ;
    if(temp == NULL) return ERROR ;
```

```
    temp->data=n; qP->rear->next=temp ;
    qP->rear = temp ;
    return OK ;
}

int front(queue q, int *v) {
    if(ISEMPTY(q)) return ERROR ;
    *v= q.front->next->data ;
    return OK ;
}

int delete(queue *qP) {
    node *temp ;
```

```
    if(ISEMPTY(*qP)) return ERROR ;
    temp = qP->front ;
    qP->front=qP->front->next ;
    free(temp) ;
    return OK ;
}
```

User Program: testQueueSR.c

```
#include <stdio.h>
#include "queueSR.h"
int main() // testQueueSR.c
{
    queue q ;
    int x , err , val ;
    char c;

    init(&q);
    printf(" 'A' for add (A 15)\n") ;
    printf(" 'D' for delete\n 'F' for front\n 'E' for e
    while((c = getchar()) != 'e' && c != 'E')
```

```
        switch(c) {
    case 'a' :
    case 'A' :
        scanf("%d",&x);
        err = add(&q,x);
                if(err) printf("The Queue is full\n"
        break;

    case 'd' :
    case 'D' :
        err = delete(&q);
                if(err) printf("The Queue is empty\n
        break;
```

```
        case 'f' :
        case 'F' :
                    err = front(q , &val) ;
                    if(err) printf("The Queue is empty\n
                    else printf("%d\n",val);
            break;

        case '\n' :
        case '\t' :
        case ' '  :
            break;

        default :
            printf("Token Unknown\n");
```

```
    }
      return 0;
}
```

Union Type

So far all our data are of single type. But it is possible to have data of any one of a few different types e.g. either of type `int` or of type `double`. The C language provides support to have data of type union.

Union Type: an Example

```
typedef union data {
                int dataI;
                double dataD;
} intDouble;
```

A variable of type `union data` or `intDouble` can store either a value of type `int` or of type `double`. The allocated space is large enough to accomodate the data of the largest size. The fields are accessed like a structure.

Note

It is necessary to attach a tag to remember the type of the actual data present. We consider a stack of type `intDouble`.

Interface File: stackU.h

```
#ifndef _STACKU_H
#define _STACKU_H

#include <stdio.h>
#include <stdlib.h>
#define ERROR 1
#define OK    0

typedef struct {   // stackU.h
          union data {
                int I;
                double D;
```

```
            } data ;
            char type;
} intDouble;

struct stack {
        intDouble data ;
        struct stack *next ;
} ;
typedef struct stack node, *stack ;

#define INIT(s) ((s)=NULL)
#define ISEMPTY(s)  ((s) == NULL)
int push(stack *, intDouble) ;
int pop(stack *) ;
```

```
int top(stack, intDouble *) ;

#endif
```

Implementation File: stackU.c

```
#include "stackU.h"

int push(stack *s, intDouble n) {  // stackU.c
    stack temp ;

    temp=(stack)malloc(sizeof(node)) ;
    if(temp == NULL) {
       printf("The STACK is full\n");
       return ERROR ;
    }
    temp->data = n;
    temp->next=*s ;
```

```
    *s = temp ;
    return OK ;
}

int pop(stack *s) {
    stack temp ;
    if(ISEMPTY(*s)) {
       printf("The STACK is empty\n");
       return ERROR ;
    }
    temp=*s;
    *s=(*s)->next ;
    free(temp) ;
    return OK ;
```

```
}

int top(stack s, intDouble *val) {
    if(ISEMPTY(s)) {
       printf("The STACK is empty\n") ;
       return ERROR ;
    }
    *val = s -> data;
    return OK ;
}
```

User Program: testStackU.c

```
#include <stdio.h>
#include "stackU.h"

int main()  // testStackU.c
{
    stack s ;
    int err ;
    intDouble d;
    char c ;

    INIT(s);
    printf(" 'Ui' for push int (Ui 15)\n 'Uf' for push
```

```
printf(" 'O' for pop\n 'T' for top \n 'E' for exit
while((c = getchar()) != 'e' && c != 'E')
        switch(c) {
   case 'u' :
   case 'U' :
                scanf("%c", &c);
                if(c == 'i' || c == 'I') {
                   scanf("%d",&d.data.I);
                   d.type = 'i';
                }
                else if(c == 'f' || c == 'F') {
                        scanf("%lf",&d.data.D);
                        d.type = 'f';
                }
```

```
                err = push(&s,d);
                break;
        case 'o' :
        case 'O' :
                err = pop(&s);
                break;
        case 't' :
        case 'T' :
                err = top(s , &d) ;
                if(!err) {
                   if(d.type == 'i') printf("%d\n",
                   if(d.type == 'f') printf("%f\n",
                }
                break;
```

```
        case '\n' :
        case '\t' :
        case ' '  :
                        break;
        default :
                        printf("Token Unknown\n");
            }
    return 0;
}
```