5. Priority Queue

30 April 2024 09:59

Queue

Linear data structure that follows First In First Out (FIFO) principle

- 2 ends: rear end, front end
- Open ended at both ends
- Elements inserted at rear end and deleted at front end

Types of queue

1. Ordinary queue
2. Priority queue: Each element associated with some priority and served according to that
   - Ascending priority queue: lower value → higher priority
   - Descending priority queue: higher value → higher priority
3. Circular queue: Last element points to first element
4. Double ended queue: Insertion/deletion possible at both ends

Applications of Priority Queue

- Implementation of heap
- Djikstra's shortest path algorithm
- Prim's algorithm
- Data compression
- OS - load balance algorithm

Unions

User defined data structure which can hold members of different datatypes.
Allows data members that are mutually exclusive to share memory
- mutually exclusive: cannot exist at the same time; either one or the other

- Efficient way of using the same memory location for different purposes
- Minimum size of union = size of largest component of union
- All fields overlap and they have zero offset.
- Used in embedded devices to conserve memory

union tagname

typedef union tagname

- used in embedded devices to conserve memory

```
union tagname
{
    int a;
    char b[10];
    double c;
} variable_name;
```

minimum memory allocated: 10 bytes

```
typedef union tagname
{

} TAG;
```

```
#include <stddef.h>
printf("Offset: %u", offsetof(union tagname, a));

TAG d1 = {50};          // you can only initialise first data member
printf("a: %d \n", d1.a);

strcpy(d1.b, "abcd");
printf("b: %s \n", d1.b);

d1.c = 4.56;
printf("c: %f \n", d1.c);
```

⚠: the attribute accessed by the dot operator determines how the bit pattern of the variable is interpreted

Nested unions

```
typedef struct test
{
    int i;
    int j;
    union nest_test
    {
        int k;
        char s[100];
    } n;
}
```

// cannot use typedef if defining inside another struct/union

> **Note: Anonymous union**
> For test, you can access k with test.n.k.
> Now say the union nest_test is anonymous w/ no variable.
> You can still access k, with test.k