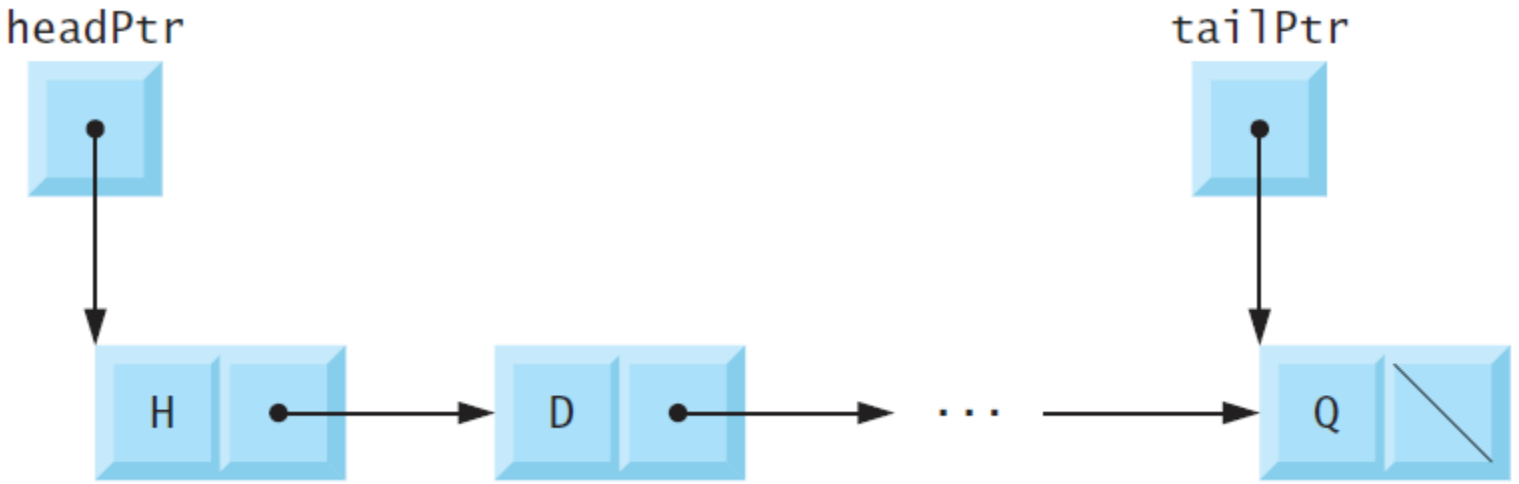
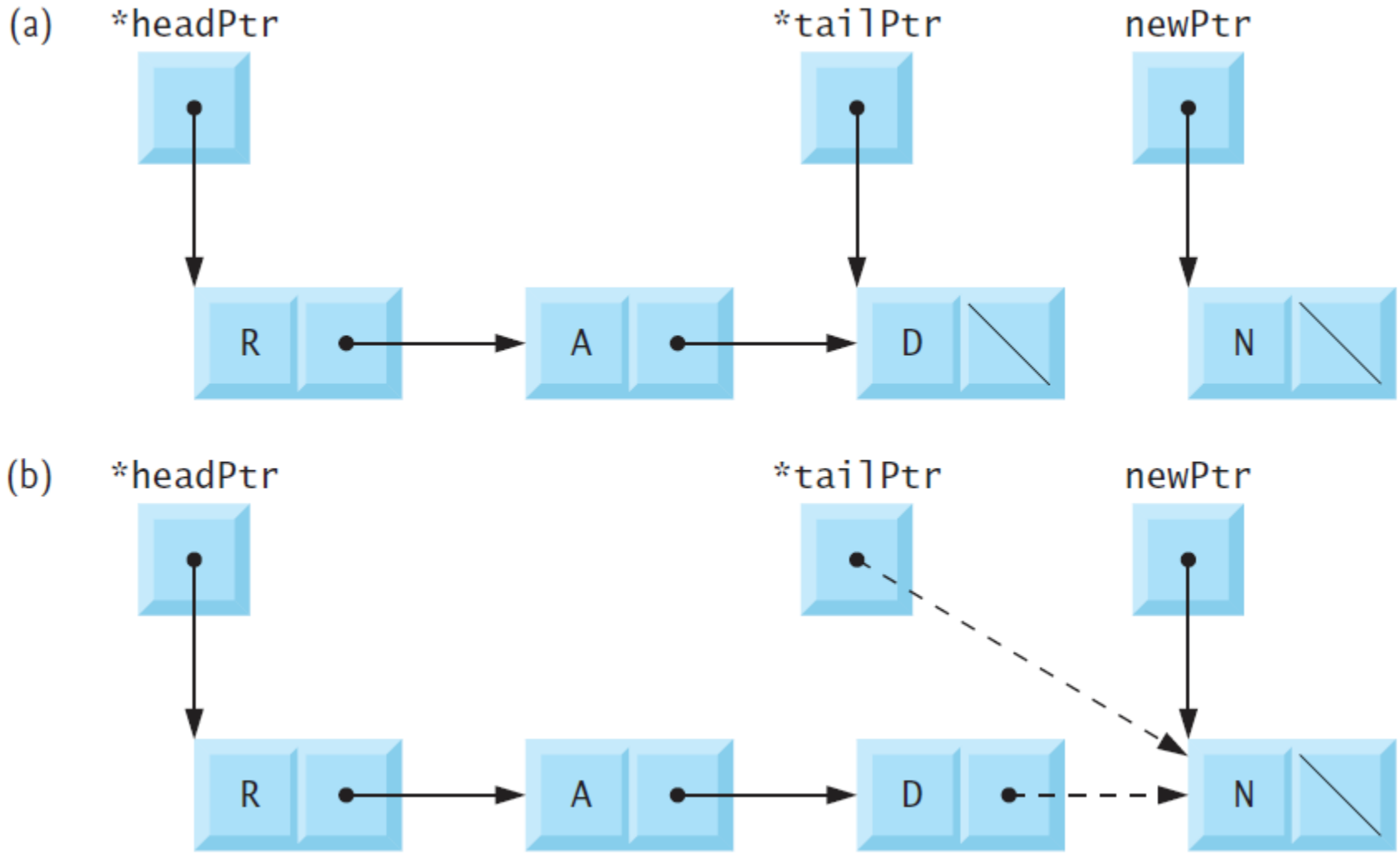
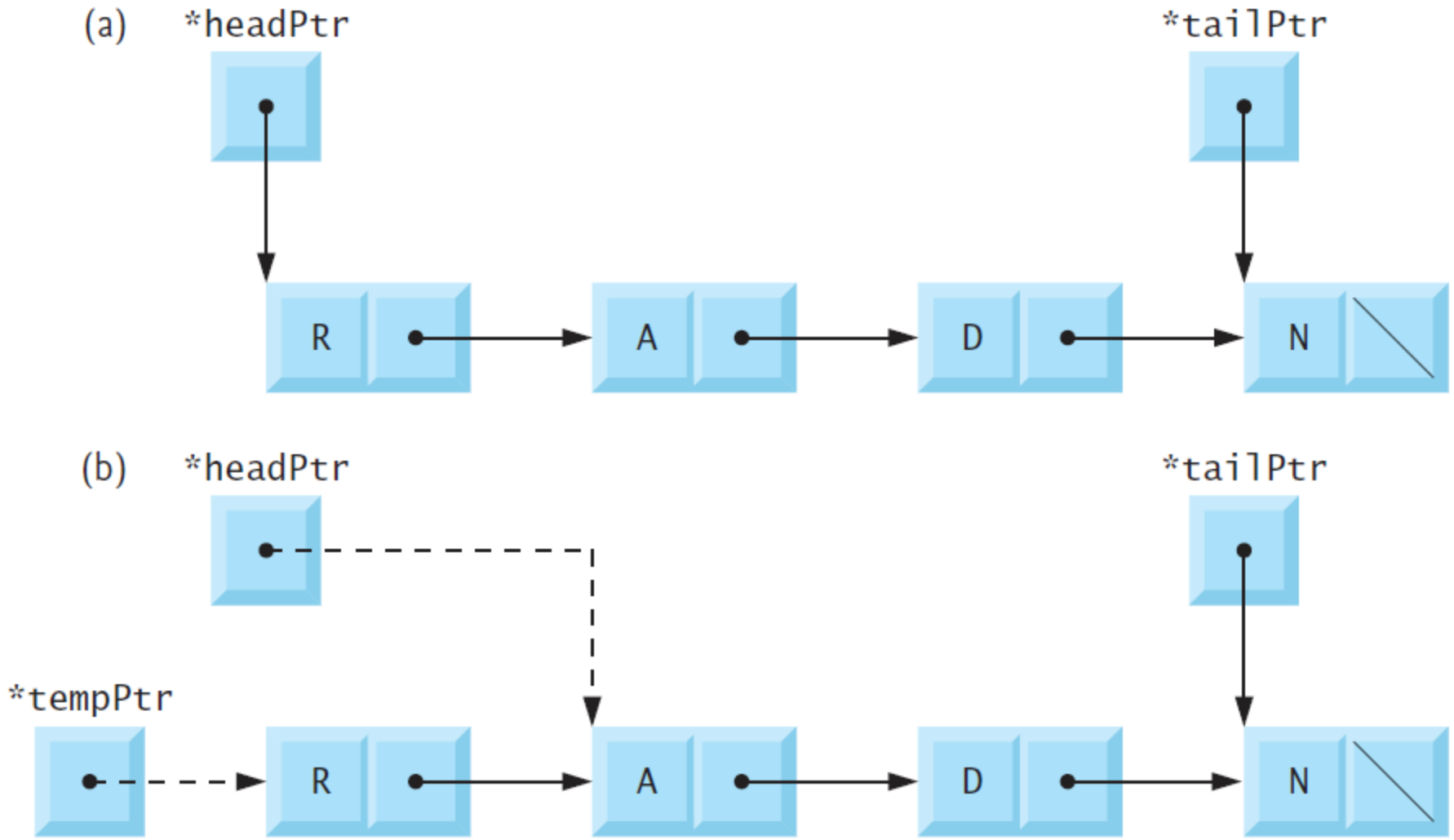
Очередь



Добавление элемента



Удаление элемента



```
#include <locale.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

enum MENU { enqueue = 1, dequeue, quit };

struct queueNode {
  char data;
  struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

void ShowMainMenu(void);
int GetMenuItem(void);
void Enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
             char value);
char Dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
bool IsEmpty(QueueNodePtr headPtr);
void PrintQueue(QueueNodePtr currentPtr);
void FreeQueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
```

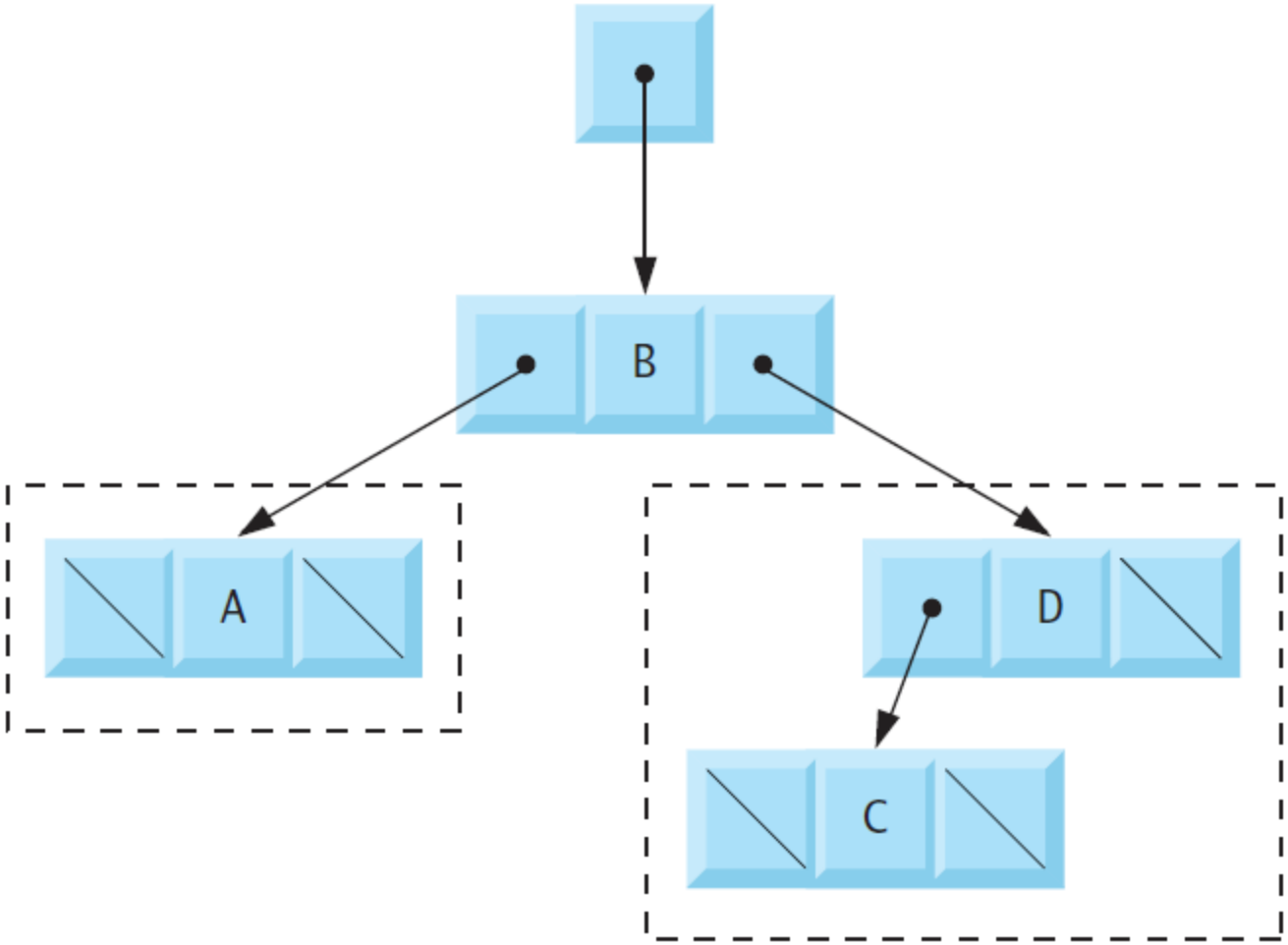
```
int main(void) {
  setlocale(LC_ALL, "RU");
  int userChoice = 0;
  QueueNodePtr headPtr = NULL;
  QueueNodePtr tailPtr = NULL;
  char item = '\0';

  for (;;) {
    ShowMainMenu();
    printf("\nВыберите пункт меню: ");
    userChoice = GetMenuItem();
    switch (userChoice) {
      case enqueue:
        printf("%s", "\nВведите символ: ");
        scanf("\n%c", &item);
        while (getchar() != '\n');
        Enqueue(&headPtr, &tailPtr, item);
        PrintQueue(headPtr);
        break;
      case dequeue:
        if (!IsEmpty(headPtr)) {
          item = Dequeue(&headPtr, &tailPtr);
          printf("\nСимвол \"%c\" удален.\n", item);
        }
        PrintQueue(headPtr);
        break;
```

```
      case quit:
        puts("\nЗавершение работы.\n");
        FreeQueue(&headPtr, &tailPtr);
        return EXIT_SUCCESS;
      default:
        puts("\nТакого пункта нет\n");
        break;
    }
  }
}

void ShowMainMenu(void) {
  printf("%d - Добавить элемент в очередь", enqueue);
  printf("\n%d - Удалить элемент из очереди", dequeue);
  printf("\n%d - Завершить работу\n", quit);
}

int GetMenuItem(void) {
  int input = 0;
  while (!scanf("%d", &input)) {
    while (getchar() != '\n')
      ;
    printf("Ошибка ввода. Введите число.\n");
  }
  while (getchar() != '\n')
    ;
  return input;
}
```

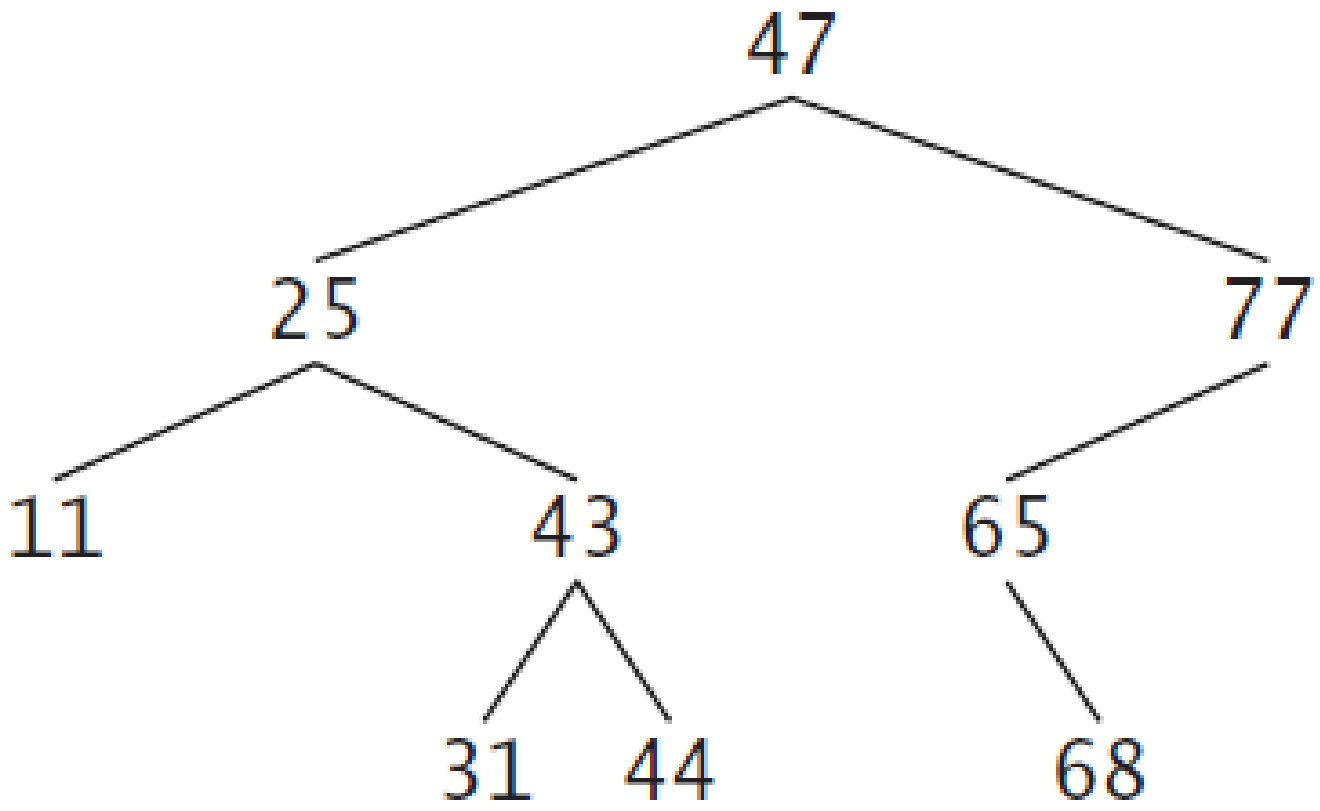
```
void Enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value) {
  QueueNodePtr newPtr = malloc(sizeof(QueueNode));
  if (newPtr != NULL) {
    newPtr->data = value;
    newPtr->nextPtr = NULL;
    if (IsEmpty(*headPtr)) {
      *headPtr = newPtr;
    }
    else {
      (*tailPtr)->nextPtr = newPtr;
    }
    *tailPtr = newPtr;
  }
  else {
    printf("Символ \"%c\" не добавлен. Ошибка выделения памяти.\n", value);
  }
}

char Dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr) {
  char value = (*headPtr)->data;
  QueueNodePtr tempPtr = *headPtr;
  *headPtr = (*headPtr)->nextPtr;
  if (*headPtr == NULL) {
    *tailPtr = NULL;
  }
  free(tempPtr);
  tempPtr = NULL;
  return value;
}
```

```
bool IsEmpty(QueueNodePtr headPtr) {
  return headPtr == NULL;
}

void PrintQueue(QueueNodePtr currentPtr) {
  if (currentPtr == NULL) {
    puts("Очередь пуста.\n");
  }
  else {
    puts("\nОчередь:");
    while (currentPtr != NULL) {
      printf("%c --> ", currentPtr->data);
      currentPtr = currentPtr->nextPtr;
    }
    puts("NULL\n");
  }
}

void FreeQueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr) {
  while (*headPtr != NULL) {
    QueueNodePtr tempPtr = *headPtr;
    *headPtr = (*headPtr)->nextPtr;
    free(tempPtr);
    tempPtr = NULL;
  }
  *tailPtr = NULL;
}
```

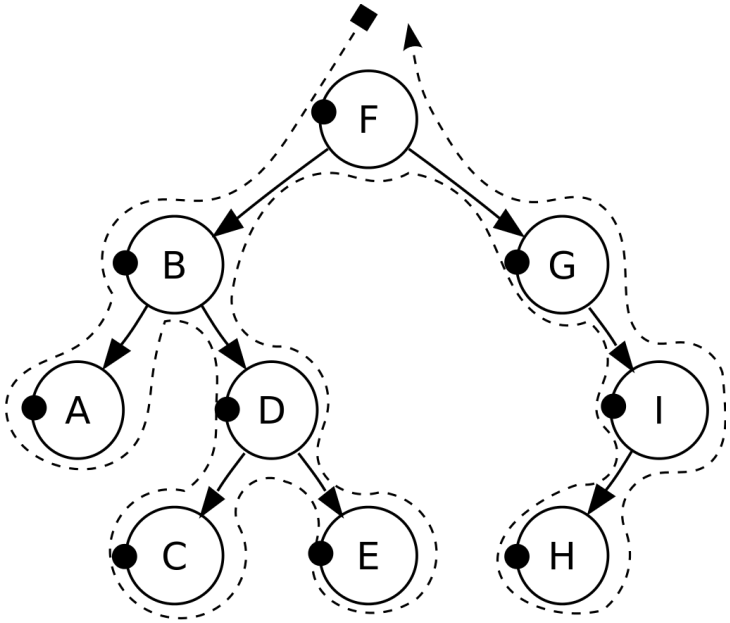
Двоичное дерево поиска



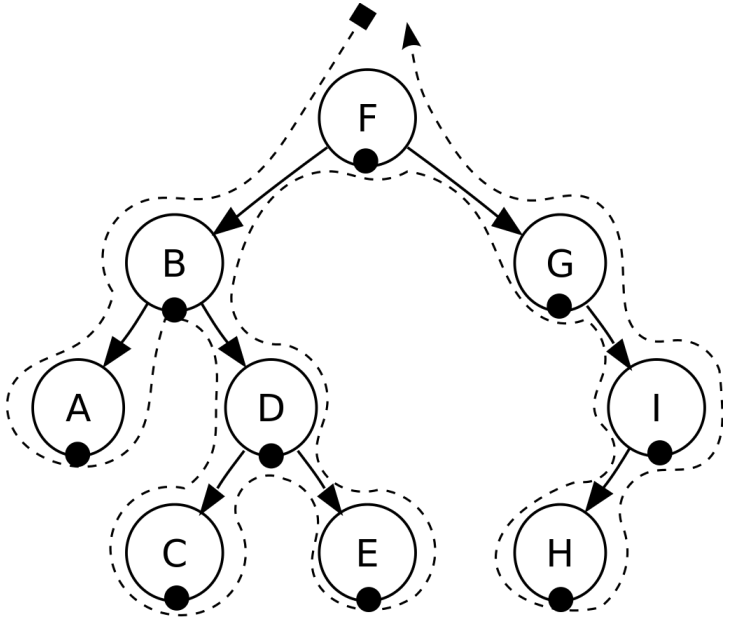
Двоичное дерево поиска



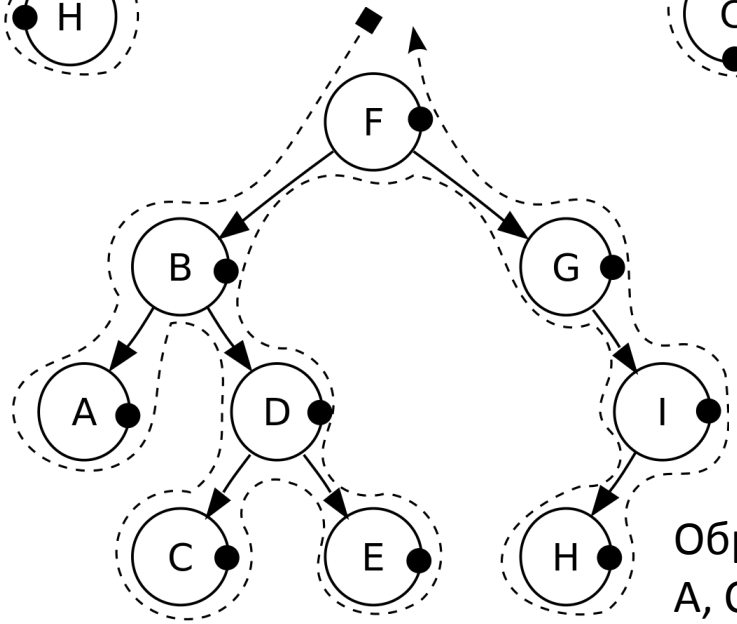
Прямой, центрированный и обратный обход



Прямой обход:
F, B, A, D, C, E, G, I, H.



Центрированный обход:
A, B, C, D, E, F, G, H, I.



Обратный обход:
A, C, E, D, B, H, I, G, F.

```
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NUMBER_OF_NODES 10
#define RANDOM_BOUND 15

enum MENU { quit };

struct treeNode {
  int data;
  struct treeNode *leftPtr;
  struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;
typedef TreeNode *TreeNodePtr;

int GetInt(void);
void InsertNode(TreeNodePtr *treePtr, int value);
void PreOrder(TreeNodePtr treePtr, int level);
void InOrder(TreeNodePtr treePtr);
void PostOrderFree(TreeNodePtr *treePtr);
```

```
int main(void) {
  setlocale(LC_ALL, "RU");
  srand(time(NULL));
  int userChoice = 0;
  TreeNodePtr rootPtr = NULL;

  do {
    puts("Узлы дерева:");
    for (int i = 1; i <= NUMBER_OF_NODES; ++i) {
      int item = rand() % RANDOM_BOUND;
      printf("%3d", item);
      InsertNode(&rootPtr, item);
    }

    puts("\n\nПрямой обход дерева:");
    PreOrder(rootPtr, 0);

    puts("\n\nЦентрированный обход дерева:");
    InOrder(rootPtr);

    puts("\n\nОбратный обход дерева при удалении:");
    PostOrderFree(&rootPtr);

    printf("\n\n0 - Завершить работу.\n");
    userChoice = GetInt();
  } while(userChoice != quit);
  return EXIT_SUCCESS;
}
```

```
int GetInt(void) {
  int input = 0;
  while (!scanf("%d", &input)) {
    while (getchar() != '\n')
      ;
    printf("Ошибка ввода. Введите число.\n");
  }
  while (getchar() != '\n')
    ;
  return input;
}
void PreOrder(TreeNodePtr treePtr, int level) {
  if (treePtr != NULL) {
    for (int i = 0; i < level; i++) {
      printf(i == level - 1 ? "|-" : "  ");
    }
    printf("%d\n", treePtr->data);
    PreOrder(treePtr->leftPtr, level + 1);
    PreOrder(treePtr->rightPtr, level + 1);
  }
}
void InOrder(TreeNodePtr treePtr) {
  if (treePtr != NULL) {
    InOrder(treePtr->leftPtr);
    printf("%3d", treePtr->data);
    InOrder(treePtr->rightPtr);
  }
}
```

```
void InsertNode(TreeNodePtr *treePtr, int value) {
  if (*treePtr == NULL) {
    *treePtr = malloc(sizeof(TreeNode));

    if (*treePtr != NULL) {
      (*treePtr)->data = value;
      (*treePtr)->leftPtr = NULL;
      (*treePtr)->rightPtr = NULL;
    }
    else {
      printf("Число %d не добавлено.
              Ошибка выделения памяти.\n", value);
    }
  }
  else {
    if (value < (*treePtr)->data) {
      InsertNode(&((*treePtr)->leftPtr), value);
    }

    else if (value > (*treePtr)->data) {
      InsertNode(&((*treePtr)->rightPtr), value);
    }
    else {
      printf("%s", "(повтор)");
    }
  }
}
```

```
void PostOrderFree(TreeNodePtr *treePtr) {
  if (*treePtr != NULL) {
    PostOrderFree(&((*treePtr)->leftPtr));
    PostOrderFree(&((*treePtr)->rightPtr));
    printf("%3d", (*treePtr)->data);
    free(*treePtr);
    *treePtr = NULL;
  }
}
```